

Ф.М. ГАФАРОВ, А.Ф. ГАЛИМЯНОВ

ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

Учебное пособие

Казань – 2018

УДК 004.032.26
ББК 32.973.2-018
Г12

*Печатается по постановлению
Редакционно-издательского совета
Института вычислительной математики и информационных технологий
Казанского (Приволжского) федерального университета;
(протокол №1 от 18 октября 2018 г.)*

Научный редактор
кандидат педагогических наук, доцент Ч.Б. Миннегалиева

Рецензенты:
кандидат физико-математических наук,
доцент кафедры теории функций и приближений КФУ **Ю.Р. Агачев;**
кандидат технических наук,
с.н.с. научно-исследовательского института АН РТ «Прикладная семиотика»
А.Р. Гатиятуллин

Гафаров Ф.М.

Г12 Параллельные вычисления: учеб. пособие /
Ф.М. Гафаров, А.Ф. Галимянов. – Казань: Изд-во Казан. ун-та, 2018. –
149 с.

Учебное пособие посвящено изложению основ параллельных вычислений. Приводятся также все необходимые вводные материалы для дальнейшего понимания.

Адресовано, в первую очередь, студентам-бакалаврам, а также магистрам направления «Информационные системы и технологии», а также широкому кругу читателей, интересующихся параллельными вычислениями.

УДК 004.032.26
ББК 32.973.2-018

© Гафаров Ф.М., Галимянов А.Ф., 2018
© Издательство Казанского университета, 2018

Оглавление

1. ОСНОВНЫЕ ПОНЯТИЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ	5
1.1. Терминология параллельных вычислений	6
2. АРХИТЕКТУРА ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ.....	11
2.1. Введение.....	11
2.2. Классификация компьютерных систем.....	12
2.3. Детализация архитектур по достижимой степени параллелизма	15
2.4. Векторно-конвейерные компьютеры	17
2.5. Вычислительные системы с распределенной памятью (мультимикропроцессоры).....	19
2.6. Параллельные компьютеры с общей памятью (микропроцессоры).....	20
2.7. Кластеры	23
2.8. Концепция GRID и метакомпьютинг	24
3. ПОСТРОЕНИЕ ОЦЕНОК ПРОИЗВОДИТЕЛЬНОСТИ И ЭФФЕКТИВНОСТИ ПАРАЛЛЕЛЬНЫХ КОМПЬЮТЕРОВ	25
3.1. Основные понятия и предположения.....	25
3.2. Построение соотношений для оценки производительности	27
3.3. Законы Амдала.....	30
3.4. Закон Густавсона - Барсиса	31
3.5. Производительность конвейерных систем	32
3.6. Масштабируемость параллельных вычислений.....	33
3.7. Верхняя граница времени выполнения параллельного алгоритма	35
3.8. Факторы, влияющие на производительность, и способы ее повышения	36
4. ПОСТРОЕНИЕ ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ: ИНЖЕНЕРНЫЙ ПОДХОД	39
4.1. Постановка задачи.....	39
4.2. Классификация алгоритмов по типу параллелизма	41
4.3. Общая схема этапов разработки параллельных алгоритмов.....	43
4.4. Декомпозиция в задачах с параллелизмом по данным	46
4.5. Блочная декомпозиция с учетом локализации подобластей.....	49
4.6. Общие рекомендации по разработке параллельных программ	53
5. THREADING.....	54
5.1. Как работает Threading	54
5.2. Создание и запуск потоков. Передача данных в поток	56
5.3. Основные свойства потоков	58
5.4. Синхронизация выполнения потоков	59
5.5. Статус выполнения потока	61
5.6. Блокировка	61
5.7. Mutex	65

5.8. Семафор.....	67
5.9. Сигнализация с помощью классов EventWaitHandle	68
6. ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ В .NET 4.0	70
6.1. Введение.....	70
6.2. Параллельный цикл For	72
6.3. Параллельный цикл ForEach.....	73
6.4. Завершение параллельных циклов.....	74
6.5. Исключения и параллельные циклы.....	76
6.6. Параллельность задач и Использование Parallel.Invoke	80
6.7. Ожидание завершения параллельных задач.....	83
6.8. Задачи продолжения.....	85
6.9. Отмена выполнения задач.....	86
6.10. Параллельный LINQ.....	89
7. MPI.....	92
7.1. Введение в MPI	92
7.2. Начало работы с MPI с помощью Visual Studio 2013	93
7.3. Основные функции MPI.....	97
7.4. MPI Send and Receive	98
7.5. Элементарные типы данных MPI	99
7.6. Коллективные коммуникации в MPI.....	101
7.7. Функции Scatter, Gather и Allgather	103
7.8. Функции MPI Reduce and Allreduce	108
7.9. Группы и коммуникаторы в MPI.....	111
8. ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ НА OPENMP.....	117
8.1. Введение в OpenMP.....	117
8.2. Основы OpenMP.....	119
8.3. Параллельные регионы.....	120
8.4. Конструкции OpenMP	122
8.5. Конструкции OpenMP для распределения работ	125
8.6. Зависимость по данным в OpenMP	133
8.7. Средства синхронизации в OpenMP	135
8.8. Расширенные возможности OpenMP	138
8.9. Отладка OpenMP кода.....	140
9. Применение Windows API в параллельных вычислениях.....	143
Литература.....	148

1. ОСНОВНЫЕ ПОНЯТИЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Параллельные вычисления (параллельная обработка) – это использование нескольких или многих вычислительных устройств для одновременного выполнения разных частей одной программы (одного проекта).

Параллельные вычисления – такой способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно. (Википедия)

Параллельные вычисления – вычисления, которые можно реализовать на многопроцессорных системах с использованием возможности одновременного выполнения многих действий, порождаемых процессом решения одной или многих задач [одного проекта]. (Словарь по кибернетике)

Основная цель параллельных вычислений – уменьшение времени решения задачи. Многие необходимые для нужд практики задачи требуется решать в реальном времени или для их решения требуется очень большой объем вычислений.

Отметим, что увеличение числа процессоров не обязательно приводит уменьшению времени решения задачи. (Если небольшую яму попытаются рыть одновременно 10 человек, то они будут только мешать друг другу.) Использование параллельной обработки данных – не единственный путь увеличить скорость вычислений. Другой подход – увеличивать мощность процессорных устройств. Ограничениями такого подхода являются:

1. Ограниченность скорости переключения. Даже при самых быстрых коммуникациях – оптических – скорость переключения не может превышать скорость света.

2. Ограниченность размеров переключателей. Чем меньше размер компонентов устройства, тем быстрее устройство может работать. Однако

существует физический предел на размер компонентов, что связано с их молекулярным и атомным строением.

3. Экономические ограничения. Для увеличения скорости процессора, плотности упаковки, числа слоев в кристалле приходится решать все усложняющиеся научные, инженерные, производственные проблемы. Вот почему каждое новое поколение процессоров дорого стоит.

Всегда найдутся большие задачи, для решения которых потребуются мощности параллельного компьютера.

Задача параллельных вычислений – создание ресурса параллелизма (получение параллельного алгоритма) в процессах решения задач и управление реализацией этого параллелизма с целью достижения наибольшей эффективности использования многопроцессорной вычислительной техники.

Получить параллельный алгоритм решения задачи можно путем распараллеливания имеющегося последовательного алгоритма или путем разработки нового параллельного алгоритма. Возможно, для осуществления распараллеливания алгоритм решения задачи придется заменить или модифицировать (например, устранить некоторые зависимости между операциями).

1.1. Терминология параллельных вычислений

Терминологию параллельных вычислений нельзя считать устоявшейся, поэтому некоторые из приведенных ниже понятий могут допускать иную трактовку.

Суперкомпьютер – вычислительная машина, значительно превосходящая по своим техническим параметрам большинство существующих компьютеров.

Кластер – группа компьютеров, объединенных в локальную вычислительную сеть и способных работать в качестве единого вычислительного ресурса.

Кластер предполагает более высокую надежность и эффективность, нежели просто локальная вычислительная сеть. Кластер использует типовые аппаратные и программные решения и поэтому имеет существенно более низкую стоимость в сравнении с другими типами параллельных вычислительных систем.

Распределённые вычисления – способ решения трудоёмких вычислительных задач с использованием нескольких компьютеров, объединённых в параллельную вычислительную систему. Слабосвязанные, гетерогенные вычислительные системы выделяют в отдельный класс распределённых систем – Grid. (Википедия)

Распределённые вычисления – технология обработки данных, в которой большая задача распределяется для выполнения между множеством компьютеров, объединённых вычислительной сетью или интернетом.

Облачные вычисления – технология обработки данных, в которой компьютерные ресурсы и мощности предоставляются как интернет-сервис.

Таким образом, распределённые и облачные вычисления являются частным случаем параллельных вычислений. Параллельные вычисления могут производиться как на одном компьютере (суперкомпьютере или многоядерном компьютере), так и на многих компьютерах. Распределённые и облачные вычисления на одном компьютере производиться не могут.

Параллельные задания – задания, допускающие одновременное (не обязательно независимое) выполнение.

Параллельный алгоритм – алгоритм, операции которого могут выполняться одновременно (не обязательно независимо); подразумевается, что в явном или неявном виде указаны одновременно выполняемые операции или множества операций. Строгое понятие параллельного алгоритма не введено.

Параллельная программа – параллельный алгоритм, записанный в некоторой системе программирования, ориентированной на вычислительные системы параллельной архитектуры.

Параллелизм по данным. Функциональный параллелизм. Если при решении некоторой задачи процессоры выполняют одинаковую последовательность вычислений, но используют разные данные, то говорят о параллелизме по данным. Например, при поиске по базе данных каждый процессор может работать со своей частью базы данных. Если процессоры выполняют разные задания одной задачи, выполняют разные функции, то говорят о функциональном параллелизме. Мы будем в основном рассматривать параллелизм по данным.

Конвейерная обработка данных операции (конвейерный параллелизм на уровне операции). Пусть операция разбита на микрооперации. Расположим микрооперации в порядке выполнения и для каждого выполнения выделим отдельную часть устройства. В первый момент времени входные данные поступают для обработки в первую часть. После выполнения первой микрооперации первая часть передает результаты своей работы второй части, а сама берет новые данные. Когда входные аргументы пройдут все этапы обработки, на выходе устройства появится результат выполнения операции. Таким образом, реализуется функциональный параллелизм. Каждая часть устройства называется ступенью конвейера, а общее число ступеней – длиной конвейера.

Реальная производительность вычислительной системы – количество операций, реально выполняемых в среднем за единицу времени.

Пиковая производительность вычислительной системы – максимальное количество операций, которое может быть выполнено системой за единицу времени.

Из определений вытекает, что реальная и пиковая производительности системы есть суммы соответственно реальных и пиковых производительностей, составляющих систему процессоров. Пиковая производительность одного процессора вычисляется как произведение $n \times f \times k$, где n – количество операций

с плавающей запятой, выполняемых за один такт, f – тактовая частота процессора, k – количество ядер в процессоре.

Пиковую производительность еще называют теоретической производительностью. Такое название подчеркивает, что на реальной программе производительность не только не превысит, но никогда и не достигнет этого порога.

Загруженность процессора на данном отрезке времени – отношение времени реальной работы процессора на данном отрезке к длине всего отрезка. Загруженность вычислительной системы, состоящей из одинаковых процессоров – среднее арифметическое загруженностей всех процессоров. Из определения следует, что загруженность p удовлетворяет условию $0 \leq p \leq 1$.

Ускорение реализации алгоритма на вычислительной системе, состоящей из одинаковых процессоров – отношение времени выполнения алгоритма на одном процессоре (на одном ядре процессора) ко времени параллельного выполнения. Ускорение зависит от выбора параллельного алгоритма и от того, насколько этот алгоритм адекватен архитектуре вычислительной системы.

Эффективность реализации алгоритма на вычислительной системе, состоящей из s одинаковых процессоров – отношение ускорения к s .

Общая (разделенная, совместно используемая) память.
Распределенная память. При параллельной обработке используются две основные модели доступа к памяти: общая, когда все процессоры напрямую связаны с одной общей памятью (но каждый процессор имеет и свою кэш-память), и распределенная, когда память физически распределена между процессорами.

Мультипроцессоры – системы с общей разделяемой памятью.

Мультикомпьютеры – системы с распределенной памятью.

Синхронизация (работы процессоров). В многопроцессорной системе необходимо координировать обмен данными между процессорами, а также, при использовании общей памяти, между процессорами и общей памятью.

Синхронизация – это согласование по времени выполнения параллельных заданий. Она включает в себя ожидание того, что выполнение задачи достигнет особой точки, называемой точкой синхронизации. После того, как все задания достигнут точки синхронизации, выполнение заданий может быть продолжено до следующей точки синхронизации. При использовании модели передачи сообщений (обычно MPI – Message Passing Interface) работу процессоров синхронизируют функции обмена данными.

Синхронизация нужна для того, чтобы согласовать обмен информацией между заданиями (между параллельно выполняемыми множествами операций). Синхронизация может привести к простоя процессора, т.к. после достижения точки синхронизации он должен ждать, пока другие задания достигнут точки синхронизации. Задержка с подачей в процессор необходимых данных ведет к простоя процессора и снижению эффективности параллельной обработки.

Зернистость. Под зерном вычислений (или тайлом) понимается множество операций алгоритма, выполняемых атомарно: вычисления, принадлежащие одному зерну, не могут прерываться синхронизацией или обменом данными, требуемыми для выполнения этих операций.

Зернистость – это мера отношения количества вычислений, сделанных параллельной задаче, к количеству пересылок данных. Мелкозернистый параллелизм – очень мало вычислений на каждую пересылку данных. Крупнозернистый параллелизм – интенсивные вычисления на каждую пересылку данных (данные пересылаются большими порциями). Чем мельче зернистость, тем больше точек синхронизации.

Поток, нить (thread), легковесный процесс – создаваемый операционной системой объект внутри процесса (процесс – выполняемое приложение с собственным виртуальным адресным пространством), который выполняет инструкции программы. Процесс может иметь один или несколько потоков, выполняемых в контексте данного процесса. Потоки (нити) позволяют осуществлять параллельное выполнение процессов и одновременное

выполнение одним процессом различных частей программы на различных процессорах.

Естественный параллелизм. Алгоритм обладает естественным параллелизмом, если его можно разбить на независимо выполняемые части.

Внутренний параллелизм. Алгоритм обладает внутренним параллелизмом, если его можно разбить на параллельно (но не обязательно независимо) выполняемые части.

Распараллеливание. Выявление (указание) операций или множеств операций последовательного алгоритма, которые могут выполняться одновременно.

Статическое распараллеливание. Распараллеливание, осуществляемое до начала выполнения алгоритма (программы).

Динамическое распараллеливание. Распараллеливание, осуществляемое во время выполнения алгоритма (программы).

Масштабируемость. Масштабируемость данного параллельного алгоритма при реализации на данной параллельной системе означает, что производительность системы пропорциональна числу содержащихся в ней процессоров (ядер). Различают понятия слабой и сильной масштабируемости. Слабая (отнюдь не в смысле «плохая») масштабируемость (weak scaling) означает линейный рост размера задачи с ростом числа процессоров при фиксированном времени выполнения алгоритма. Сильная масштабируемость (strong scaling) означает линейный рост ускорения с ростом числа процессоров при фиксированном размере задачи.

2. АРХИТЕКТУРА ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

2.1. Введение

Рассмотрим сначала процесс подготовки параллельного решения задачи на многопроцессорной системе как отдельный этап подготовки описания параллельного алгоритма на некотором языке блок-схем и/или графов.

Написание параллельной программы рассматривается как завершающий этап, обеспечивающий эффективную реализацию задуманного алгоритма, возможно, с учетом его конкретных особенностей.

Другими словами, при описании параллельных алгоритмов не обязательно знать конкретный язык программирования, на котором будет реализована программа. Однако надо хорошо знать особенности вычислительной системы, на которой будет реализован алгоритм (типы используемых вычислительных узлов, производительность и др.). Если у разработчика есть выбор, можно поставить задачу построения наиболее эффективного параллельного алгоритма, подобрав типы вычислителей, наиболее полно реализующие его особенности. Для этого необходимо ясно представлять потенциальные возможности различных архитектур.

Таким образом, изучение возможных типов архитектур, характеристик и способов организации вычислительной системы, на которой предполагается реализация разрабатываемого параллельного алгоритма, является необходимым этапом. Рассмотрим наиболее популярные архитектуры в том минимальном объеме, который может потребоваться при разработке параллельного алгоритма.

2.2. Классификация компьютерных систем

Существуют различные классификации, преследующие разные цели. При разработке параллельного алгоритма наиболее важно знать тип оперативной памяти, т.к. она определяет способ взаимодействия между частями параллельной программы. В зависимости от организации подсистем оперативной памяти параллельные компьютеры можно разделить на следующие два класса.

Системы с разделяемой памятью (мультимикропроцессоры), у которых имеется одна виртуальная память, а все процессоры имеют одинаковый доступ к данным и командам, хранящимся в этой памяти (uniform memory access или UMA). По этому принципу строятся векторные параллельные процессоры

(parallel vector processor или PVP) и симметричные мультипроцессоры (symmetric multiprocessor или SMP).

Системы с распределенной памятью (мультимышнытеры), у которых каждый процессор имеет свою локальную оперативную память, а у других процессоров доступ к этой памяти отсутствует.

При работе на компьютерe с распределенной памятью необходимо создавать копии исходных данных на каждом процессоре. В случае системы с разделяемой памятью достаточно один раз задать соответствующую структуру данных и разместить ее в оперативной памяти.

Указанные два типа организации памяти могут быть реализованы в различных архитектурах. Рассмотрим различные классификации параллельных компьютеров, указывая там, где это имеет значение, способ организации оперативной памяти.

Исторически наиболее ранней является классификация *М. Флинна* (1966). Классификация основана на понятии *потока*, под которым понимается последовательность команд или данных, обрабатываемых процессором. На основе числа потоков команд и потоков данных выделяют четыре класса архитектур:

- SISD (Single Instruction stream/Single Data stream) - один поток команд и один поток данных;
- SIMD (Single Instruction stream/Multiple Data stream) - один поток команд и множество потоков данных;
- MISD (Multiple Instruction stream/Single Data stream) - множество потоков команд и один поток данных;
- MIMD (Multiple Instruction stream/Multiple Data stream) - множество потоков команд и множество потоков данных.

В настоящее время подавляющее число «серьезных» компьютеров реализуется в классе MIMD-архитектур. При этом рассматривают следующие основные подклассы.

Векторно-конвейерные компьютеры, в которых используется набор векторных команд, обеспечивающих выполнение операций с массивами независимых данных за один такт. Типичным представителем данного направления является линия «классических» векторно-конвейерных компьютеров CRAY.

Массово-параллельные (чаще называемые также **массивно-параллельные**) компьютеры с *распределенной* памятью. В данном случае микропроцессоры, имеющие каждый свою локальную память, соединяются посредством некоторой коммуникационной среды. Достоинство этой архитектуры - возможность наращивать производительности путем добавления процессоров. Недостаток - большие накладные расходы на межпроцессорное взаимодействие.

Симметричные мультипроцессоры (SMP) состоят из совокупности процессоров, имеющих разделяемую общую память с единым адресным пространством и функционирующих под управлением одной операционной системы. Недостаток - число процессоров, имеющих доступ к общей памяти, нельзя сделать большим. Существует предел наращивания числа процессоров, превышение которого ведет к быстрому росту потерь на межпроцессорный обмен данными.

Кластеры образуются из вычислительных модулей любого из рассмотренных выше типов, объединенных системой связи или посредством разделяемой внешней памяти. Могут использоваться как специализированные, так и универсальные сетевые технологии. Это направление, по существу, является комбинацией предыдущих трех.

Наиболее важным при разработке параллельного алгоритма является деление на компьютеры с общей и распределенной памятью. Для компьютеров с общей памятью пользователю не нужно заботиться о распределении данных, достаточно предусмотреть лишь затраты на выбор необходимых данных из этой памяти. При реализации параллельного алгоритма на компьютерах с

распределенной памятью необходимо продумать рациональную, с точки зрения потерь на обмен данными, схему их размещения. Далее дадим более подробную характеристику каждого из указанных выше подклассов компьютеров.

2.3. Детализация архитектур по достижимой степени параллелизма

Выше мы рассмотрели основные классы параллельных компьютеров, отличия между которыми следует учитывать в первую очередь при построении параллельных алгоритмов. Поскольку подавляющее число архитектур реализуется в классе MIMD, требуется более детальная классификация, которая, кроме прочего, позволяла бы также давать оценку достижимой степени параллелизма.

Одна из таких систематизаций MIMD-компьютеров дана *Р. Хокни* [2]. Основная идея классификации состоит в том, что множественный поток команд может быть обработан либо по конвейерной схеме в режиме разделения времени, либо каждый поток обрабатывается своим устройством.

В соответствии с этим различают следующие MIMD - компьютеры:

- конвейерные;
- переключаемые (с общей памятью и распределенной памятью);
- сети, реализованные в виде: регулярной решетки, гиперкуба, иерархической структуры и изменяемой конфигурации.

Следующая классификация [2] *Т. Фенга* позволяет также строить оценки достижимой степени параллелизма. Она основана на двух характеристиках:

- число n бит в машинном слове, обрабатываемых параллельно;
- число слов m , обрабатываемых одновременно вычислительной системой.

Произведение $P=m*n$, определяющее интегральную характеристику параллельности архитектуры, называют **максимальной степенью параллелизма** вычислительной системы. Введение этой **единой числовой метрики** для всех типов компьютеров позволяет сравнивать любые два

компьютера между собой. Однако в данном случае не делается акцент на том, за счет чего компьютер может одновременно обрабатывать более одного слова.

С точки зрения указанной классификации возможны следующие варианты построения компьютера:

- разрядно-последовательные, пословно-последовательные ($n=1$, $m=1$);
- разрядно-параллельные, пословно-последовательные ($n>1$, $m=1$);
- разрядно-последовательные, пословно-параллельные ($n=1$, $m>1$);
- разрядно-параллельные, пословно-параллельные ($n>1$, $m>1$).

Подавляющее большинство вычислительных систем принадлежит к этому, последнему, классу.

Классификация *В. Хендлера* [2]. В основе этой классификации явное описание параллельной и конвейерной обработки. При этом различают три уровня обработки данных:

- уровень выполнения программы;
- уровень выполнения команд;
- уровень битовой обработки.

На каждом уровне допускается возможность конвейерной обработки. Таким образом, в общем случае каждый компьютер может быть охарактеризован следующими шестью числами:

k - число процессоров;

k' - глубина макроконвейера;

d - число АЛУ в каждом процессоре;

d' - глубина конвейера из функциональных устройств АЛУ;

w - число разрядов в слове, обрабатываемых в АЛУ параллельно;

w' - число ступеней в конвейере функциональных устройств каждого АЛУ.

Имеет место связь классификации *Хендлера* с классификацией *Фенга*: для получения максимальной степени параллелизма в смысле *Фенга* необходимо

вычислить произведение указанных выше шести величин.

В классификации Д. Скилликорна [2] архитектуру любого компьютера предлагается рассматривать как абстрактную структуру, состоящую из четырех компонентов:

- *процессор команд* (IP - Instruction Procesor) - интерпретатор команд;
- *процессор данных* (DP - Data Procesor) - устройство обработки данных;
- *устройство памяти* (IM - Instruction Memory, DM - Data Memory);
- *переключатель* - абстрактное устройство, обеспечивающее связь между процессорами и памятью.

Рассматриваются четыре типа переключателей:

- 1-1 связывает пару функциональных устройств;
- n-n - реализует попарную связь каждого устройства из одного множества с соответствующим ему устройством из другого множества;
- 1-n - соединяет одно выделенное устройство со всеми функциональными устройствами из некоторого набора;
- n*n - каждое функциональное устройство одного множества может быть связано с любым устройством из некоторого набора.

Заметим, что приведенные в настоящем разделе типы классификаций претендуют на более высокий уровень формализации количественных оценок параллелизма и поэтому могут быть полезными при проведении исследований, связанных с применением моделей вычислительных систем достаточно высокого уровня абстрактности.

2.4. Векторно-конвейерные компьютеры

Появление термина *суперкомпьютер* связано с созданием в середине шестидесятых годов фирмой CDC (Сеймуром Крэм) высокопроизводительного компьютера с новой *векторной* архитектурой. Основная идея, положенная в основу этой архитектуры, заключалась в

распараллеливании процесса обработки данных, когда одна и та же операция применяется одновременно к массиву (вектору) значений. Эта идея оказалась плодотворной и нашла воплощение на разных уровнях функционирования компьютера.

Классическим представителем мира суперкомпьютеров является первый векторно-конвейерный компьютер Cray-1 (1976). Основные особенности архитектуры этого класса компьютеров следующие.

- *Конвейеризация выполнения команд.*
- *Независимость функциональных устройств*, т.е. несколько операций могут выполняться одновременно.
- *Векторная обработка* (набор данных обрабатывается одной командой).
- *Зацепление функциональных устройств* (выполнение нескольких векторных операций в режиме «макроконвейера»).
- *Многопроцессорная обработка* (наличие независимых процессоров позволяет выполнять несколько независимых программ).

Эффективность векторно-конвейерных компьютеров существенным образом зависит от наличия одинаковых и независимых операций. В качестве примера рассмотрим несколько фрагментов вычислений в виде блок-схем, показанных на рисунке 2.1, а, б, в.

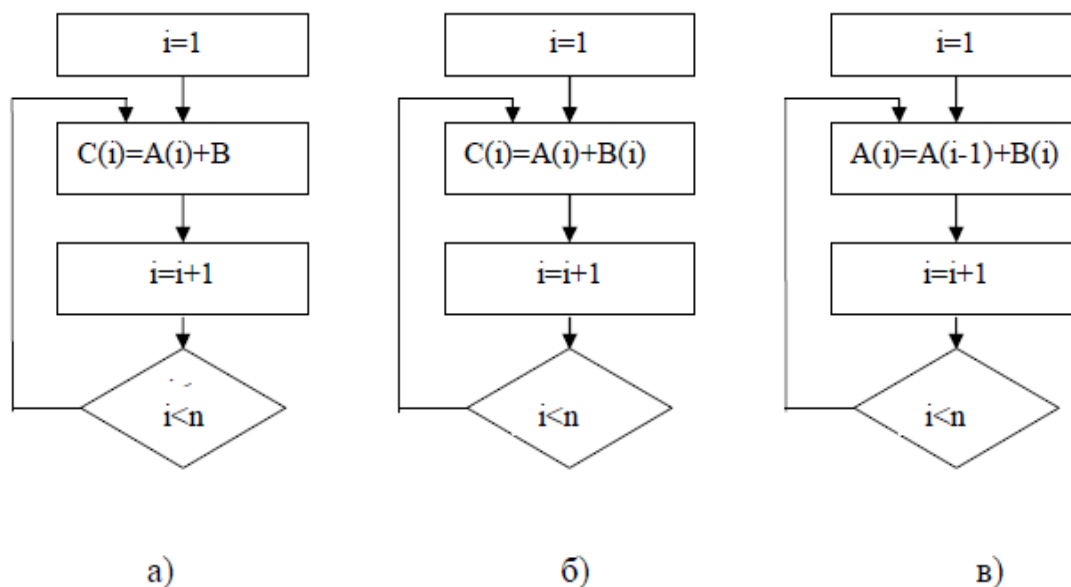


Рис. 2.1 Примеры векторизуемых и невекторизуемых алгоритмов

Поскольку в системе команд векторно-конвейерных компьютеров обычно есть векторные команды, в которых аргументы могут быть как скалярами, так и векторами, векторизация фрагментов, показанных на рис. 2.1, а и б, не вызовет проблем. В то же время фрагмент, показанный на рис. 2.1, в, невозможно векторизовать, поскольку вычисление i -го элемента массива A не может начаться, пока не будет вычислен предыдущий элемент. В данном примере имеет место *зависимость между операциями*, которая будет препятствовать векторизации. Это надо иметь в виду при выполнении программы на компьютере векторно-конвейерной архитектуры.

В качестве примера предположим, что половина некоторой программы - это сугубо последовательные вычисления, которые нельзя векторизовать. Тогда, даже в случае мгновенного выполнения второй половины программы за счет идеальной векторизации, ускорения работы всей программы более чем в два раза мы не получим.

2.5. Вычислительные системы с распределенной памятью (мультимикрокомпьютеры)

Как уже указывалось выше, вычислительные узлы этого класса (массивно-параллельных) компьютеров объединяются друг с другом

посредством коммуникационной среды. Каждый узел имеет один или несколько процессоров и свою собственную локальную память. Распределенность памяти означает, что каждый процессор имеет непосредственный доступ только к локальной памяти своего узла. Доступ к памяти других узлов осуществляется посредством либо специально проектируемой для данной вычислительной системы, либо стандартной коммуникационной среды.

Преимущества этой архитектуры - низкое значение отношения цена/производительность и возможность практически неограниченно наращивать число процессоров. Различия компьютеров данного класса сводятся к различиям в организации коммуникационной среды. Известны архитектуры, в которых процессоры расположены в узлах прямоугольной решетки. Иногда взаимодействие идет через иерархическую систему коммутаторов, обеспечивающих возможность связи каждого узла с каждым. Используется также топология трехмерного тора, т.е. каждый узел имеет шесть

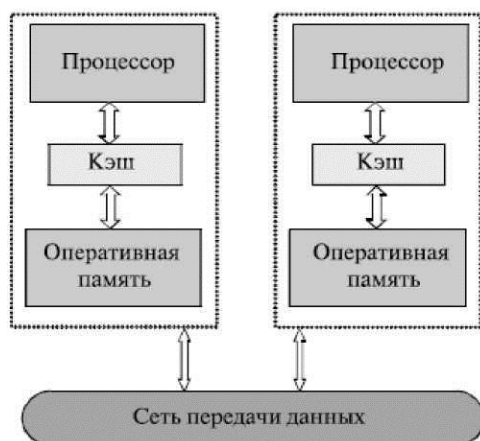


Рис. 2.2 Архитектура многопроцессорных систем с распределенной памятью

непосредственных соседей вне зависимости от того, где он расположен.

На рис. 2.2 показана общая схема связей основных элементов системы в архитектуре многопроцессорных систем с распределенной памятью.

2.6. Параллельные компьютеры с общей памятью (мультипроцессоры)

Организация параллельных вычислений для компьютеров этого класса значительно проще, чем для систем с распределенной памятью. В данном случае не надо думать о распределении массивов. Однако компьютеры этого класса имеют небольшое число процессоров и очень высокую стоимость. Поэтому обычно используются различные решения, позволяющие увеличить число процессоров, но сохранить возможность работы в рамках единого адресного пространства.

В частности, общая память может быть физически распределенной, однако все процессоры имеют доступ к памяти любого процессора. Достигается это применением специальных программно-аппаратных средств. Основная проблема, которую при этом решают - обеспечение когерентности кэш-памяти отдельных процессоров. Реализация мероприятий по обеспечению когерентности кэшей позволяет значительно увеличить число параллельно работающих процессоров по сравнению с SMP-компьютером. Такой подход именуется неоднородным доступом к памяти (non-uniform memory access или NUMA). Среди систем с таким типом памяти выделяют:

- системы, в которых для представления данных используется только локальная кэш-память процессоров (cache-only memory architecture или COMA);
- системы, в которых обеспечивается когерентность локальных кэшей разных процессоров (cache-coherent NUMA или CC-NUMA);
- системы, в которых обеспечивается общий доступ к локальной памяти разных процессоров без поддержки на аппаратном уровне когерентности кэша (non-cache coherent NUMA или NCC-NUMA).

На рис. 2.3. приведены некоторые типовые схемы связей элементов в мультипроцессорных системах.

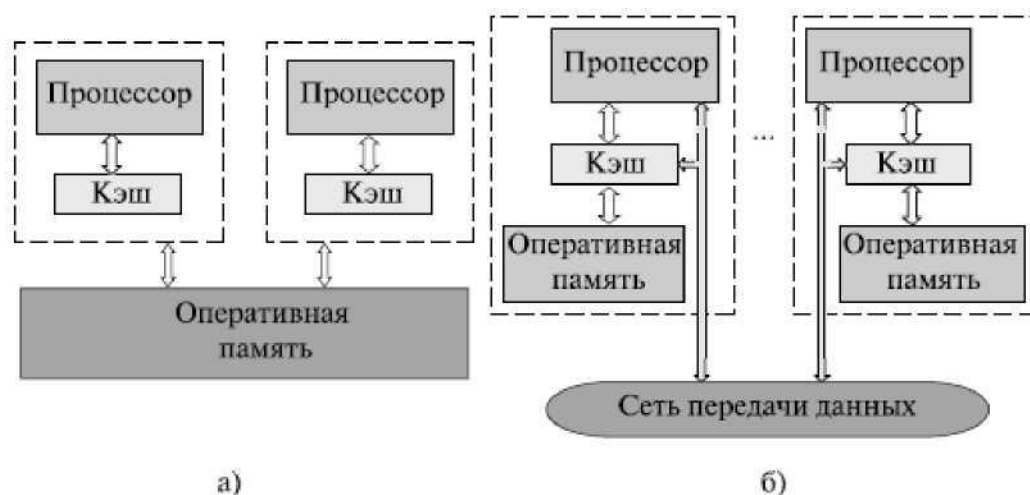


Рис. 2.3 Архитектура многопроцессорных систем с общей (разделяемой) памятью: системы с однородным (а) и неоднородным (б) доступом к памяти

Использование распределенной общей памяти (distributed shared memory или DSM) упрощает проблемы создания мультипроцессоров (известны примеры систем с несколькими тысячами процессоров). Однако при построении параллельных алгоритмов в данном случае необходимо учитывать, что время доступа к локальной и удаленной памяти может различаться на несколько порядков. Для обеспечения эффективности алгоритма в этом случае следует в явном виде планировать распределение данных и схему обмена данными между процессорами таким образом, чтобы минимизировать обращения к удаленной памяти.

Есть существенные различия векторных и массивно-параллельных архитектур. В векторной программе явно выполняются операции над всеми элементами регистра, в параллельной программе каждый из процессоров выполняет более или менее синхронно машинные команды, оперируя со своими собственными регистрами. В обоих случаях действия выполняются одновременно, однако каждый из процессоров параллельной ЭВМ может реализовывать свой алгоритм, отличающийся от алгоритмов других процессоров.

Указанное отличие является весьма существенным. Справедливо следующее утверждение: алгоритм, который можно векторизовать, можно и

распараллелить. Обратное утверждение не всегда верно. Например, для не векторизуемого фрагмента алгоритма, показанного на рис. 2.1, в, нетрудно организовать конвейерную схему вычислений на массивно-параллельном компьютере.

2.7. Кластеры

Кластеры являются одним из направлений развития компьютеров с массовым параллелизмом. Кластерные проекты связаны с появлением на рынке недорогих микропроцессоров и коммуникационных решений. В результате появилась реальная возможность создавать установки «суперкомпьютерного» класса из составных частей массового производства.

Один из первых кластерных проектов - Beowulf-кластеры. Первый кластер был собран в 1994 г. в центре NASA Goddard Space Flight Center (GSFC). Он включал 16 процессоров Intel 486DX4/100 МГц. На каждом узле было установлено по 16 Мбайт оперативной памяти и сетевые карты Ethernet. Чуть позже был собран кластер TheHIVE (Highly-parallrl Integrated Virtual Environment). Этот кластер включал 332 процессора и два выделенных хост-компьютера. Все узлы кластера работали под управлением Red Hat Linux.

В настоящее время известно огромное количество кластерных решений. Одно из существенных различий состоит в используемой сетевой технологии. При использовании массовых сетевых технологий, обладающих низкой стоимостью, как правило, возникают большие накладные расходы на передачу сообщений.

Для характеристики сетей в кластерных системах используют два параметра: латентность и пропускную способность. *Латентность* - это время начальной задержки при посылке сообщений. *Пропускная способность сети* определяется скоростью передачи информации по каналам связи. Если в параллельном алгоритме много коротких сообщений, то критической характеристикой является латентность. Если передача сообщений организована большими порциями, то более важной является пропускная способность

каналов связи. Указанные две характеристики могут оказывать огромное влияние на эффективность исполнения кода.

Если в компьютере не поддерживается возможность *асинхронной отправки сообщений* на фоне вычислений, то возникают неизбежные при этом накладные расходы, связанные с ожиданием полного завершения взаимодействия параллельных процессов. Для повышения эффективности параллельной обработки на кластере необходимо добиваться *равномерной загрузки всех процессоров*. Если этого нет, то часть процессоров будет простаивать. В случае, когда вычислительная система неоднородна (гетерогенна), балансировка загрузки процессоров становится крайне трудной задачей.

В заключение еще раз зададимся вопросом: чем же все-таки кластеры отличаются от других компьютерных систем? Следуя [2] приведем следующее утверждение «Отличие понятия кластера от сети компьютеров (network of workstations) состоит в том, что для построения локальной компьютерной сети, как правило, используют более простые сети передачи данных, компьютеры сети обычно более рассредоточены, а пользователи могут применять их для выполнения каких-либо дополнительных работ». Впрочем, эта граница все чаще оказывается в значительной степени «размытой», в связи с бурным ростом пропускной способности сетей передачи данных.

2.8. Концепция GRID и метакомпьютинг

В принципе, любые вычислительные устройства можно считать параллельной вычислительной системой, если они работают одновременно и их можно использовать для решения одной задачи. Под это определение попадают и компьютеры в сети Интернет. Интернет можно рассматривать как самый мощный кластер - метакомпьютер. Процесс организации вычислений в такой вычислительной системе - *метакомпьютинг*. В отличие от традиционного компьютера метакомпьютер имеет некоторые, присущие только ему особенности:

- огромные вычислительные ресурсы (число процессоров, объем

памяти и др.);

- присущая ему от природы распределенность ресурсов;
- возможность динамического изменения конфигурации (подключений);
- неоднородность;
- объединение ресурсов различных организаций, с различающейся политикой доступа (по принадлежности).

Из указанных особенностей следует, что метакомпьютер - это не только и не столько сами вычислительные устройства, сколько инфраструктура. В данном случае в комплексе должны рассматриваться модели программирования, распределение и диспетчеризация заданий, организация доступа, интерфейс с пользователем, безопасность, надежность, политика администрирования, средства и технологии распределенного хранения данных и др.

Необходимо подчеркнуть, что развитие идей метакомпьютинга связано с актуальными проблемами переработки огромных объемов информации. Характерной задачей является создание информационной инфраструктуры для поддержки экспериментов в физике высоких энергий в Европейском центре ядерных исследований (CERN). Для обработки большого объема данных целесообразно создание иерархической распределенной системы, включающей ряд связанных высокоскоростными телекоммуникационными каналами центров различного уровня. При этом центры могут быть удалены друг от друга на значительные расстояния. Основой для построения инфраструктуры указанного типа являются GRID-технологии [2].

3. ПОСТРОЕНИЕ ОЦЕНОК ПРОИЗВОДИТЕЛЬНОСТИ И ЭФФЕКТИВНОСТИ ПАРАЛЛЕЛЬНЫХ КОМПЬЮТЕРОВ

3.1. Основные понятия и предположения

Несмотря на большое обилие различных архитектур, существует лишь два способа параллельной обработки данных: собственно, параллелизм и

конвейерность. В компьютере обычно реализуются все основные типы команд: скалярные, векторные и конвейерные. Команда, у которой все аргументы скалярные величины, называется скалярной командой. Если хотя бы один аргумент вектор, команда называется векторной. Соответственно в составе компьютера могут быть скалярные, векторные и конвейерные устройства. Введем необходимые определения и предположения, касающиеся оценки производительности вычислительных систем.

Предполагается, что система состоит из набора функциональных устройств (ФУ). Результат предыдущего срабатывания ФУ может сохраняться в нем только до момента очередного срабатывания. ФУ не может одновременно выполнять операцию и сохранять результат, т.е. не имеет собственной памяти. ФУ называется простым, если никакая последующая операция не может начаться раньше, чем предыдущая. Конвейерное ФУ состоит из цепочки простых ФУ, которые называют элементарными. Очередная операция считается выполненной после прохождения всех элементарных ФУ (ступеней конвейера).

Пусть время выполнения одной операции τ . Тогда за время T может быть выполнено приблизительно T/τ операций (здесь и во многих случаях далее для простоты мы не учитываем, что следует брать только целую часть результата деления). Время τ реализации одной операции называют стоимостью операции, а сумму стоимостей всех операций T - стоимостью работы. Минимально возможное время выполнения алгоритма определяется длиной критического пути. Загруженностью устройства - ρ называют отношение стоимости реально выполненной работы к максимально возможной стоимости. Показатель эффективности одного процессора - количество операций, запускаемых за один такт процессора - IPC (instructions per cycle). Общая вычислительная мощность многопроцессорной системы оценивается пиковой производительностью, определяемой как максимальное количество операций, которое может быть выполнено системой за единицу времени при отсутствии потерь времени на

связи между ФУ. Единица измерения производительности - Flops (одна вещественная операция в секунду).

Пиковая производительность многопроцессорной системы определяется как количество функциональных устройств, предназначенных для выполнения операций с плавающей точкой (равное числу IPC), умноженное на частоту работы процессора и на число процессоров. Например, для компьютера с двумя устройствами с плавающей точкой и частотой 500 МГц пиковая производительность равна 1000 Mflops (1 Gflops). Эффективность использования других функциональных устройств (целочисленная арифметика, обращение к памяти и др.) выявляется путем сравнения реально достижимой на тестах производительности с пиковой.

Реальная производительность - это количество операций, реально выполняемых в среднем в единицу времени. Реальная производительность обычно существенно меньше пиковой. Превышение пиковой производительности над реальной характеризует, насколько данная архитектура приспособлена к решению конкретной задачи. Отношение реальной производительности к пиковой называется эффективностью реализации задачи на данном конкретном компьютере.

Эффективность реализации программы повышается в том случае, когда возрастает относительная загруженность АЛУ. Для этого необходимо устранять узкие места, которые обычно связаны с временем обращения к памяти и временем пересылки данных. Если на большинстве задач эффективность работы компьютера более 0,5, ситуацию можно считать хорошей [2].

3.2. Построение соотношений для оценки производительности

Если s устройств системы имеют пиковые производительности π_1, \dots, π_s и работают с загруженностями p_1, \dots, p_s , то реальная производительность системы выражается формулой [2]

$$r = p\pi, \quad (3.1)$$

где $\pi = \pi_1 + \dots + \pi_s$, а p - загруженность системы, определяемая как

$$p = \sum_{i=1}^s \alpha_i p_i, \alpha_i = \frac{\pi_i}{\sum_{j=1}^s \pi_j} \quad (3.2)$$

Из (3.1) видно, что для достижения наибольшей реальной производительности системы при фиксированном числе устройств необходимо обеспечить наиболее полную ее загруженность. Дальнейшее повышение производительности достигается увеличением числа устройств.

Ускорение реализации алгоритма на вычислительной системе из s устройств определяется как [2]

$$R_s = \frac{r}{\pi_s} \quad (3.3)$$

где π_s - пиковая производительность самого быстроедействующего устройства системы. Это означает, что наибольшее ускорение - s системы из s устройств может достигаться только в случае, когда все устройства системы имеют одинаковые пиковые производительности и полностью загружены. Реальное ускорение для однородных вычислительных систем, имеющих одинаковую производительность устройств, часто определяют также как отношение времени решения задачи на одном процессоре - T_1 к времени T_s решения той же задачи на системе из s таких же процессоров:

$$R = \frac{T_1}{T_s} \quad (3.4)$$

Это соотношение можно получить также из (3.3) с учетом (3.1), т.к. при одинаковой производительности устройств $p = T_1/T_s \cdot s$, а $\pi = \pi_s s$.

Отношение реального ускорения к числу используемых процессоров s :

$$E_s = \frac{R}{s} = \frac{T_1}{T_s \cdot s} \quad (3.5)$$

называют эффективностью системы. Второе равенство в (3.5) показывает, что при одинаковой производительности устройств эффективность системы совпадает со значением загруженности системы. Далее всюду, где имеет место этот случай, под загруженностью системы подразумевается эффективность и применяется обозначение, принятое в (3.5). Наилучшие показатели ускорения и эффективности - соответственно $R=s$, $E_s = 1$.

Для анализа производительности вычислительных систем, в которых имеют место направленные связи между устройствами, воспользуемся моделью в виде ориентированного графа, в котором вершины обозначают устройства, а дуги - связи между ними [2]. Предположим, дуга графа системы идет из i -го устройства в j -е. Поскольку результат i -го устройства является аргументом j -го, количество операций, выполняемых j -м устройством, не может более, чем на 1, отличаться от количества операций, реализованных i -м устройством:

$$N_i - 1 \leq N_j \leq N_i + 1 \quad (3.6)$$

Допустим, связный граф содержит q дуг. Если k -е устройство за время T выполнило N_k операций, а l -е - N_l операций, то из (3.6) вытекает, что

$$N_l - q \leq N_k \leq N_l + q$$

для любых $k, l, 1 \leq k, l \leq s$.

Перенумеруем устройства так, что $\pi_1 \leq \pi_2 \leq \dots \leq \pi_s$. Тогда в соответствии с последним неравенством можно записать

$$(N_1 - q)s + q \leq \sum_{i=1}^s N_i \leq (N_1 + q)s - q \quad (3.7)$$

Разделив все части неравенств (3.7) на T с учетом того, что

$$\frac{1}{T} \sum_{i=1}^s N_i = r, \frac{N_1}{T} = \pi_1,$$

указанные неравенства можно переписать в виде

$$\pi_1 s - \frac{q(s-1)}{T} \leq r \leq \pi_1 s + \frac{q(s-1)}{T} \quad (3.8)$$

Слагаемые $q(s-1)/T$ в неравенствах (3.8) при увеличении T стремятся к нулю. Это означает, что для системы из s устройств с пиковыми производительностями π_1, \dots, π_s описываемой связным графом, максимальная производительность r_{max} определяется как

$$r_{max} = s \min_{1 \leq i \leq s} \pi_i \quad (3.9)$$

3.3. Законы Амдала

Из (3.7). (3.8) вытекают важные следствия [2]:

1. Загруженность системы не превосходит

$$p_{max} = \frac{s \min_{1 \leq i \leq s} \pi_i}{\sum_{i=1}^s \pi_i} \quad (3.10)$$

2. Ускорение системы не превосходит

$$R_{max} = \frac{s \min_{1 \leq i \leq s} \pi_i}{\max_{1 \leq i \leq s} \pi_i} \quad (3.11)$$

3.1-й закон Амдала. Производительность вычислительной системы, состоящей из связанных между собой устройств, определяется самым непроизводительным устройством.

4. Асимптотическая производительность системы максимальна, если все устройства имеют одинаковые пиковые производительности.

Центральное значение для оценки производительности многопроцессорных вычислительных систем имеет [2].

2-й закон Амдала:

Пусть система состоит из s одинаковых устройств, а n операций из общего числа операций алгоритма N могут выполняться только последовательно, тогда максимально возможное ускорение равно

$$R = \frac{s}{\beta \cdot s + (1 - \beta)}, \quad (3.12)$$

где $\beta = \frac{n}{N}$.

Покажем это. Если пиковые производительности всех устройств одинаковы и равны l , в соответствии с (3.1) - (3.3) ускорение определяется как

$$R = \sum_{i=1}^s p_i \quad (3.13)$$

Загруженность устройства, на котором выполняется последовательная часть программы, равна единице. Загруженности остальных устройств

$$p_i = \frac{(1 - \beta)N/s}{\beta N + (\beta - 1)N/s}, i = \overline{2, s}.$$

Следовательно, в соответствии с (3.13)

$$R = 1 + \sum_{i=2}^s \frac{(1 - \beta)N/s}{\beta N + (\beta - 1)N/s} = \frac{s}{\beta s + (1 - \beta)}.$$

Формула Амдала используется для прогноза возможного ускорения. Например, в случае, когда половина операций не поддаются распараллеливанию, максимально достижимое ускорение в случае использования 2 процессоров в соответствии с (3.12) составит около 1,33, для 10 процессоров - менее 1,82, а для 100 процессоров - около 1,98. В данном примере наиболее «узким» местом является сам алгоритм решения задачи, а основные усилия должны быть направлены на поиск другой формулировки задачи, допускающей более высокую степень параллелизма.

3.4. Закон Густавсона - Барсиса

Оценку максимально достижимого ускорения параллельного алгоритма можно построить также исходя из имеющейся доли последовательных расчетов, задаваемой в виде [3]:

$$g = \frac{\tau_n}{\tau_n + \tau_{N-n}/s}. \quad (3.14)$$

где τ_n и τ_{N-n} - время, необходимое для выполнения последовательной и параллельной частей соответственно.

С учетом введенных обозначений время решения задачи на одном и s процессорах соответственно

$$T_1 = \tau_n + \tau_{N-n}, T_{s1} = \tau_n + \tau_{N-n}/s. \quad (3.15)$$

С другой стороны, из соотношения (3.14) для величины g можно записать:

$$\tau_n = g \left(\tau_n + \frac{\tau_{N-n}}{s} \right), \tau_{N-n} = (1 - g)s \left(\tau_n + \frac{\tau_{N-n}}{s} \right). \quad (3.16)$$

С учетом (3.4), (3.15) и (3.16) получаем оценку для ускорения

$$R = \frac{T_1}{T_s} = \frac{\tau_n + \tau_{N-n}}{\tau_n + \tau_{N-n}/s} = \frac{(g + (1-g)s)(\tau_n + \frac{\tau_{N-n}}{s})}{\tau_n + \tau_{N-n}/s} = g + (1 - g)s. \quad (3.17)$$

Оценку (3.17) называют законом Густавсона - Барсиса. Нетрудно заметить, что эту оценку можно также переписать в виде:

$$R = \frac{T_1}{T_s} = s + (1 - s)g. \quad (3.18)$$

3.5. Производительность конвейерных систем

Если ФУ конвейерного типа, то операция разбивается на последовательность микроопераций. Каждую микрооперацию выделяют в отдельную часть устройства и располагают их в порядке выполнения так, чтобы входные аргументы прошли через все ступени конвейера. Рассмотрим возникающие при этом особенности оценки производительности устройства [2].

Предположим, что конвейерное устройство состоит из l ступеней, срабатывающих за один такт. Тогда, например, для сложения двух векторов из n элементов потребуется $l+n-l$ тактов. Если при этом используются также векторные команды, то потребуется (возможно, несколько) дополнительных

тактов σ для их инициализации. Эта величина учитывает также возможные пропуски тактов выдачи результатов на выходе конвейера, вследствие необходимости выполнения вспомогательных операций, связанных с организацией конвейера.

С использованием введенных обозначений запишем соотношение для оценки производительности конвейера:

$$E = \frac{n}{t} = \frac{n}{[(\sigma+l+n-1)\tau]} = \frac{1}{\left[\tau + (\sigma+l-1)\frac{\tau}{n}\right]}, \quad (3.19)$$

где τ - время такта работы компьютера.

Обычно вычислительные системы строятся с использованием одновременно всех типов устройств: скалярных, векторных конвейерных. В частности, первый векторно-конвейерный компьютер Cray-1 (пиковая производительность 160 Mflops) имел 12 конвейерных функциональных устройств, причем все функциональные устройства могли работать одновременно и независимо друг от друга.

3.6. Масштабируемость параллельных вычислений

Параллельный алгоритм называют масштабируемым (scalable), если при росте числа процессоров он обеспечивает увеличение ускорения при сохранении эффективности использования процессоров. Для характеристики свойств масштабируемости оценивают накладные расходы (время T_0) на организацию взаимодействия процессоров, синхронизацию параллельных вычислений и т.п.:

$$T_0 = sT_s - T_1 \quad (3.20)$$

где T_s , T_1 - те же, что и в (3.4).

Используя введенные обозначения, соотношения для времени параллельного решения задачи и соответствующего ускорения можно представить в виде

$$T_s = \frac{T_1 + T_0}{s}, \quad (3.21)$$

$$R_s = \frac{T_1}{T_s} = \frac{sT_1}{T_1 + T_0}. \quad (3.22)$$

Соответственно эффективность использования s процессоров

$$p_s = \frac{R_s}{s} = \frac{T_1}{T_1 + T_0} = \frac{1}{1 + T_0/T_1}. \quad (3.23)$$

Из (3.23) следует, что если время решения последовательной задачи фиксировано ($T_1 = \text{const}$), то при росте числа процессоров эффективность может убывать лишь за счет роста накладных расходов T_0 .

Если число процессоров фиксировано, эффективность их использования, как правило, растет при повышении времени (сложности) решаемой задачи T_1 . Связано это с тем, что при росте сложности задачи накладные расходы T_0 обычно растут медленнее, чем объем вычислений T_1 . Для характеристики свойства сохранения эффективности при увеличении числа процессоров и повышении сложности решаемых задач строят так называемую функцию изоэффективности. Рассмотрим схему ее построения.

Пусть задан желаемый уровень эффективности выполняемых вычислений:

$$p_s = \text{const}.$$

Из выражения для эффективности (3.23) можно записать

$$\frac{T_0}{T_1} = \frac{1 - p_s}{p_s}.$$

Или

$$T_1 = KT_0, \text{ где } K = \frac{p_s}{1 - p_s}.$$

Из последнего равенства видно, что эффективность характеризуется коэффициентом K . Следовательно, если построить функцию вида

$$N = F(K, s),$$

то для заданного фиксированного уровня эффективности K каждому числу процессоров s можно поставить в соответствие требуемый уровень сложности - N и наоборот. При рассмотрении конкретных вычислительных алгоритмов построение функции изоэффективности позволяет выявить пути совершенствования параллельных алгоритмов.

Для построения этих функций удобно использовать закон Густавсона - Барсиса. Эффективность использования s процессоров в соответствии с этим законом выражается в виде

$$E_s = \frac{R}{s} = 1 + \frac{(1-s)}{s} g.$$

При заданном фиксированном $p_s = \text{const}$ с использованием этого равенства можно построить аналитическое соотношение для функции изоэффективности в следующем виде:

$$g = F(E_s, s).$$

Такая форма может оказаться более удобной в случае, когда известна доля времени на проведение последовательных расчетов в выполняемых параллельных вычислениях.

3.7. Верхняя граница времени выполнения параллельного алгоритма

Для любого количества используемых процессоров - s справедлива следующая верхняя оценка для времени выполнения параллельного алгоритма

$$T_s < T_\infty + T_1/s \quad (3.24)$$

Действительно, пусть H_∞ есть расписание для достижения минимально возможного времени выполнения T_∞ . Для каждой итерации $\tau, 0 < \tau < T_\infty$ выполнения расписания H_∞ обозначим через n_τ количество операций, выполняемых в ходе итерации τ . Расписание выполнения алгоритма с

использованием s процессоров может быть построено следующим образом. Выполнение алгоритма разделим на T_∞ шагов; на каждом шаге τ следует выполнить все n_τ операций, которые выполнялись на итерации τ расписания H_∞ . Эти операции могут быть выполнены не более чем за n_τ/s итераций при использовании s процессоров. Как результат, время выполнения алгоритма T_s может быть оценено следующим образом:

$$T_s = \sum_{T=T_1}^{T_\infty} \left\lceil \frac{n_\tau}{s} \right\rceil < \sum_{T=T_1}^{T_\infty} \left\lceil \frac{n_\tau}{s} + 1 \right\rceil = \frac{T_1}{s} + T_\infty, \quad (3.25)$$

где $\lceil * \rceil$ - означает операцию округления до целого числа в сторону увеличения.

Приведенная схема рассуждений, по существу, дает практический способ построения расписания параллельного алгоритма. Первоначально может быть построено расписание без учета ограниченности числа используемых процессоров (расписание для **паракомпьютера**). Затем, в соответствии с описанной выше схемой, может быть построено расписание для конкретного количества процессоров.

3.8. Факторы, влияющие на производительность, и способы ее повышения

Для того чтобы правильно интерпретировать достигнутые показатели ускорения и эффективности при решении конкретной задачи на параллельном компьютере, надо ясно представлять все факторы, которые влияют на производительность. Известно, что на компьютере с огромной пиковой производительностью можно не получить ускорения или даже получить замедление счета по сравнению с обычным персональным компьютером. Перечислим факторы, которые влияют на производительность.

Архитектура процессоров. Например, если решается задача, в которой отсутствуют массивы данных, элементы которых могут обрабатываться одновременно, а каждая следующая операция может выполняться лишь после

завершения предыдущей, тогда применение мощного векторного суперкомпьютера ничего не даст.

Память и системная шина, соединяющая микропроцессоры с памятью. Пропускная способность системной шины оказывает большое влияние на показатели ускорения и эффективности, особенно если в задаче много обменов данными между процессорами.

Кэш-память. Большое значение имеет ее объем, частота работы, организация отображения основной памяти в кэш-память. Эффективность кэш-памяти зависит от типа задачи, в частности, от рабочего множества адресов и типа обращений, которые связаны с локальностью вычислений и локальностью использования данных. Наиболее характерным примером конструкции, обладающей свойством локальности, является цикл. В циклах на каждой итерации выполняются одни и те же команды над данными, которые обычно получены на предшествующей операции. Существенное ускорение выполнения циклов достигается путем его размещения его данных в кэш-памяти. Если объема кэш-памяти не хватает, задействуется следующий уровень иерархии памяти, и т.д. Именно кэш-память чаще всего оказывает наиболее существенное влияние на характеристики программ вообще, и распараллеливаемой задачи в частности.

Коммутационные сети. Они определяют накладные расходы - время задержки передачи сообщения. Оно зависит от латентности (начальной задержки при посылке сообщений) и длины передаваемого сообщения. На практике о величине латентности судят по времени передачи пакета нулевой длины.

Программное обеспечение. Операционная система, драйвера сетевых устройств, программы, обеспечивающие сетевой интерфейс нижнего уровня, библиотека передачи сообщений (MPI), компиляторы оказывают огромное влияние на производительность параллельного компьютера. В настоящем курсе

лекций эти вопросы не затрагиваются. Достаточно подробное рассмотрение этих вопросов можно найти в учебных пособиях, посвященных параллельному программированию [3, 7].

Повышение производительности обычно достигается за счет увеличения параллельно работающих процессоров. При этом основная проблема - организация связи между процессорами. Конечно, самый простой способ коммутации процессоров - использование общей шины. Однако в таких системах даже небольшое увеличение числа процессоров, подключаемых к общей шине, делает ее узким местом.

Применяются различные способы преодоления этой проблемы, основанные на использовании различных схем коммутации. Если число процессоров и модулей памяти, для связи между которыми используются коммутаторы, велико, в схеме также возможны большие задержки. Уменьшение задержек достигается путем подбора наиболее подходящей топологии сети, обеспечивающей уменьшение средней длины пути между двумя узлами системы. Среднюю длину пути можно уменьшить, применяя вместо простой линейки схему в виде кольца или гиперкуба.

Как уже указывалось, Интернет можно рассматривать как самый большой компьютер с распределенной памятью. В рамках этой архитектуры сокращение расходов на взаимодействие (обмен данными) параллельно работающих процессоров является наиболее острой проблемой. Подходящими для реализации в распределенной сети компьютеров являются задачи, которые могут быть представлены в виде фрагментов, не требующих частых обменов данными.

Отдельное, стоящее несколько особняком, направление повышения производительности компьютеров - применение специализированных процессоров. В этом случае высокая производительность достигается за счет использования особенностей конкретных алгоритмов. При этом компьютер

теряет в гибкости и универсальности. Широко используемыми, например, являются спецпроцессоры для аппаратной поддержки реализации быстрого преобразования Фурье. Этот путь оправдан, когда суперкомпьютер создается для решения специального класса часто решаемых задач, например, для подготовки ежедневного прогноза погоды.

4. ПОСТРОЕНИЕ ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ: ИНЖЕНЕРНЫЙ ПОДХОД

4.1. Постановка задачи

Известно, что перенос последовательной программы на параллельную ЭВМ без ее существенной переработки, как правило, не приводит к ускорению вычислений. Усилия, затрачиваемые на эту переработку, в значительной степени зависят от типа решаемой задачи. Для того чтобы построить эффективный параллельный алгоритм, строго говоря, следует провести анализ графа алгоритма и решить задачу отображения так, как это сформулировано в разделе 3.2. Решение такой задачи оптимизации на графах требует значительных усилий и высокой квалификации.

На практике разработку параллельного алгоритма обычно осуществляет специалист, работающий в некоторой предметной области, не всегда владеющий методами дискретной оптимизации. С другой стороны, строгое решение этой задачи требуется далеко не всегда. Обычно ограничения, связанные с типовым набором доступных архитектур, все равно вынуждают исследователя находить некоторое приемлемое для него решение, руководствуясь не вполне строгими, но проверенными на практике приемами и правилами.

В частности, если в конкретной задаче элементы некоторого массива исходных данных могут обрабатываться независимо друг от друга, то эти правила обычно очевидны и позволяют строить весьма эффективные параллельные алгоритмы. В этом случае задача переработки может свестись к

разбиению массива исходных данных на фрагменты, обработка которых ведется независимо на различных процессорах. Ясно, что при этом должна обеспечиваться равномерная загрузка процессоров, с учетом их, возможно, различной производительности.

Эффективность программы в этом случае зависит от соотношения временных затрат на проведение вычислений на фрагментах исходных данных и пересылку данных (накладные расходы) [2,3]. По мере увеличения числа (а значит уменьшения размеров) фрагментов данных, объем вычислений на каждом фрагменте уменьшается. При этом накладные расходы могут оставаться почти прежними, например, вследствие большой латентности (связанной с потерями на передачу сообщения нулевой длины) коммуникационной среды.

Иногда используется следующий простой способ построения эффективной параллельной программы, совмещенный с этапом ее отладки. Размеры фрагментов массива исходных данных уменьшают (соответственно увеличивают число параллельно работающих процессоров) до тех пор, пока имеет место почти линейное ускорение. Если же при очередном увеличении числа процессоров линейного ускорения не происходит, это означает, что накладные расходы стали заметными и дальнейшее распараллеливание по данным приведет к недостаточной загрузке процессоров. Этот подход обсуждался в работе [7].

Совокупность методов и приемов распараллеливания, не требующих строгого решения задачи отображения графа алгоритма на граф вычислительной системы, будем называть инженерным подходом. В настоящем разделе в рамках этого подхода рассматриваются некоторые правила и приемы построения параллельных алгоритмов, выработанные на основе опыта и здравого смысла. Успешность применения этих методов в значительной степени будет зависеть от соответствия структуры построенного параллельного

алгоритма типу его внутреннего параллелизма. Поэтому начнем с рассмотрения классификации алгоритмов по этому признаку.

4.2. Классификация алгоритмов по типу параллелизма

Способность алгоритма к распараллеливанию потенциально связана с одним из двух (или одновременно с обоими) внутренних свойств, которые характеризуются как параллелизм задач (message passing) и параллелизм данных (data parallel). Если алгоритм основан на параллелизме задач, вычислительная задача разбивается на несколько, относительно самостоятельных подзадач, каждая из которых загружается в "свой" процессор. Каждая подзадача реализуется независимо, но использует общие данные и/или обменивается результатами своей работы с другими подзадачами. Для реализации такого алгоритма на многопроцессорной системе необходимо выявлять независимые подзадачи, которые могут выполняться параллельно. Часто это оказывается далеко не очевидной и весьма трудной задачей. Методика решения этой задачи будет рассмотрена в следующем разделе.

При наличии в алгоритме свойства параллелизма данных, одна операция может выполняться сразу над всеми элементами массива данных. В этом случае различные фрагменты массива могут обрабатываться независимо на разных процессорах. Для алгоритмов этого типа распределение данных между процессорами обычно осуществляется до выполнения задачи на ЭВМ. Построение алгоритма, обладающего свойством параллелизма данных, и подбор подходящей архитектуры компьютера для него могут выполняться с использованием достаточно простых методик, не требующих применения сложного математического аппарата.

Для того чтобы в полной мере использовать структурные свойства алгоритма, необходимо прежде всего выявить, к какому типу он относится. Ниже приводится общая классификация алгоритмов, с точки зрения типа параллелизма, заимствованная из работы [10].

1. Алгоритмы, использующие параллелизм данных (Data Parallelism).

Этот тип параллелизма характерен для численных алгоритмов обработки, имеющих дело с большими массивами, представляемыми, например, в виде векторов и матриц. Простейшим примером такой задачи является, например, процедура перемножения двух матриц.

2. Алгоритмы с распределением данных (Data Partitioning). Это разновидность параллелизма данных, при котором пространство данных может быть разделено на непересекающиеся области, с каждой из которых связаны независимые процессы, оперирующие каждый со своими данными. Требуется лишь редкий обмен между этими процессами.

3. Релаксационные алгоритмы (Relaxed Algorithm). Алгоритм может быть представлен в виде независимых процессов без синхронизации связи между ними, но процессоры должны иметь доступ к общим данным.

4. Алгоритмы с синхронизацией итераций (Synchronous Iteration). Многие из стандартных численных итерационных параллельных алгоритмов требуют синхронизации в конце каждой итерации, заключающейся в том, что разрешение на начало следующей итерации дается после того, как все процессоры завершили предыдущую итерацию.

5. Самовоспроизводящиеся задачи (Replicated Workers). Для задач этого класса создается и поддерживается центральный пул (хранилище) похожих вычислительных задач. Параллельно реализуемые процессы осуществляют выбор задач из пула, выполнение требуемых вычислений и добавление новых задач к пулу. Вычисления заканчиваются, когда пул пуст. Эта технология характерна для исследований графа или дерева.

6. Конвейерные вычисления (Pipelined Computation). Этот тип вычислений характерен для процессов, которые могут быть представлены в виде некоторой регулярной структуры, например, в виде кольца или двумерной

сети. Каждый процесс, находящийся в узле этой структуры, реализует определенную фазу вычислений.

Нетрудно заметить, что некоторые алгоритмы из приведенного списка обладают явно выраженными свойствами параллелизма задач или параллелизма по данным. Вместе с тем ряд алгоритмов в той или иной мере обладают обоими указанными свойствами. Это следует учитывать при выборе способа и схемы декомпозиции задачи на подзадачи. Далее рассматривается основанная на разумных предположениях схема этапов построения параллельного алгоритма, основанного на декомпозиции данных.

4.3. Общая схема этапов разработки параллельных алгоритмов

В учебном пособии [3] описана технология подготовки параллельных приложений в виде следующих этапов:

1. Декомпозиция задачи на подзадачи, которые реализуются независимо.
2. Определение для сформированного набора подзадач информационных взаимодействий.
3. Масштабирование подзадач, определение количества процессоров.
4. Определение архитектуры системы, закрепление подзадач за процессорами, составление расписания.

После выполнения указанных этапов и оценки качества параллельного алгоритма (ускорения, эффективности, масштабируемости) может оказаться необходимым повторение некоторых (или всех) этапов [3]. Если в результате ряда попыток желаемые показатели качества не достигаются, следует проанализировать и, возможно, изменить математическую постановку задачи с целью построения новой вычислительной схемы.

Следует заметить, что указанная последовательность этапов носит условный характер. Часто, приступая к разработке параллельного алгоритма, пользователь ориентируется на конкретную вычислительную систему, в

частности, может быть известно возможное число доступных процессоров. Ясно, что на этапе декомпозиции по данным следует использовать эту информацию для выбора числа областей, определяющих число подзадач.

Если точное число процессоров неизвестно, но заданы границы доступного решающего поля, можно начать с масштабирования базового набора задач, а затем выполнить декомпозицию и выявление связей по информации. Другими словами, в приведенной общей схеме необходимым является лишь содержание этапов, в то время как сами этапы могут выполняться в любой последовательности, притом любой из них может оказаться как начальным, так и завершающим. На рис. 4.1 показана возможная схема взаимосвязи типовых этапов разработки алгоритмов параллельных вычислений.



Рис. 4.1 Общая схема взаимосвязи этапов разработки
параллельных алгоритмов

Если базовые подзадачи определены, установление информационных зависимостей между ними обычно не вызывает больших затруднений. При проведении анализа информационных зависимостей между подзадачами следует различать:

- локальные (на соседних процессорах) и глобальные (в которых принимают участие все процессоры) схемы передачи данных;
- структурные (соответствующие типовым топологиям коммуникаций) и произвольные способы взаимодействия;
- статические (задаваемые на этапе проектирования) или динамические (определяемые в ходе выполняемых вычислений);
- синхронные (следующая операция выполняется после выполнения предыдущей операции всеми процессорами) и асинхронные способы взаимодействия (процессы могут не дожидаться полного завершения действий по передаче данных)

Если количество подзадач (областей данных) отличается от числа процессоров, то необходимо выполнить масштабирование параллельного алгоритма. Для сокращения количества подзадач укрупняют области исходных данных, притом в первую очередь объединяют области, для которых соответствующие подзадачи обладают высокой степенью информационной взаимозависимости. Если число подзадач меньше числа доступных процессоров, выполняют декомпозицию. Масштабирование облегчается, если правила агрегации и декомпозиции параметрически зависят от числа процессоров.

Распределение подзадач между процессорами очевидно, если количество областей данных совпадает с числом имеющихся процессоров, а топология сети передачи данных - полный граф (все процессоры связаны между собой). Если это не так, подзадачи, имеющие информационные взаимодействия, целесообразно размещать на процессорах, между которыми существуют прямые линии передачи данных. Требование минимизации информационных обменов между процессорами может вступить в противоречие с условием равномерной загрузки. Решение вопросов балансировки вычислительной нагрузки значительно усложняется, если схема вычислений изменяется в ходе решения задачи. При этом необходимо перераспределение базовых подзадач

между процессорами (динамическая балансировка) в ходе выполнения программы.

Центральной проблемой, как уже неоднократно указывалось выше, является выделение базовых подзадач на этапе декомпозиции. Эта проблема имеет много аспектов, в следующем разделе кратко рассматриваются лишь некоторые важнейшие.

Описанная выше схема этапов может использоваться также и для построения параллельного алгоритма, который характеризуется параллелизмом задач. При этом содержание этапов может существенно отличаться. В частности, центральной проблемой в этом случае является выявление взаимно независимых операторов, которые могут выполняться параллельно и независимо.

4.4. Декомпозиция в задачах с параллелизмом по данным

Способ разделения вычислений на независимые части зависит от того, насколько полно решаемая задача обладает свойством декомпозируемости по данным, определяемого местом алгоритма в классификации, приведенной в разделе 4.1. Если задача допускает реализацию в классе алгоритмов с распределением данных (Data Partitioning), распараллеливание на подзадачи существенно облегчается. В данном случае одна операция или совокупность операций выполняются над всеми элементами массива данных, а задача сводится к разбиению массива исходных данных на фрагменты, обработка которых ведется независимо на различных процессорах. При этом обычно предъявляются требования обеспечить:

- примерно равный объем вычислений в выделяемых подзадачах;
- минимальный информационный обмен данными между процессорами.

Рассмотрим выполнение этих требований при различных условиях.

Простейший и наиболее благоприятный с точки зрения организации параллельных вычислений случай, когда вся область исходных данных задачи может быть разделена на непересекающиеся области любых размеров, а вычисления в каждой области могут вестись независимо. Ясно, что в этом случае задача декомпозиции чрезвычайно проста: необходимо всю область разбить на подобласти, число которых равно числу доступных процессоров, а размеры подобластей подобрать так, чтобы обеспечить их равномерную загруженность, с учетом производительности каждого.

С точки зрения организации вычислений обычно более удобной является декомпозиция на области, с границами в виде прямых линий и плоскостей. На рис. 4.2 приведены примеры наиболее широко используемых регулярных структур базовых подзадач, при декомпозиции по данным.

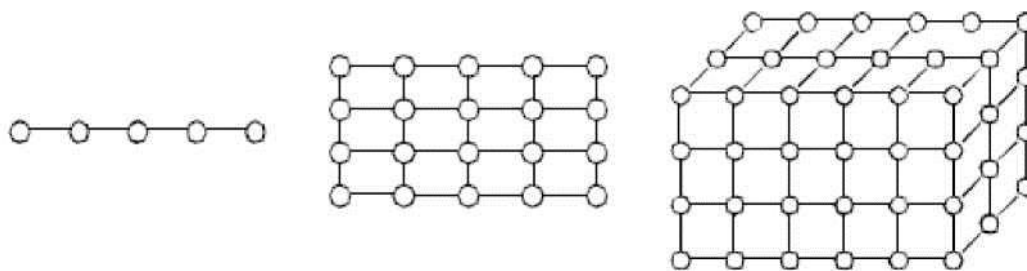


Рис. 4.2 Регулярные одно-, двух- и трехмерные структуры базовых подзадач после декомпозиции данных

Для большинства практических задач при декомпозиции по данным вычисления в каждой области не могут быть полностью независимыми. В частности, после каждой итерации (проведения вычислений во всех точках области) возникает потребность обмена результатами вычислений на границах соседних областей. Это, например, характерно для большинства сеточных методов, в которых для вычисления значения функции в некотором узле используются ее значения в нескольких соседних узлах. В этом случае выполнение указанных выше требований: сбалансированность загрузки процессоров и минимизация информационных обменов, зависит не только от размеров подобластей, но также и от их формы.

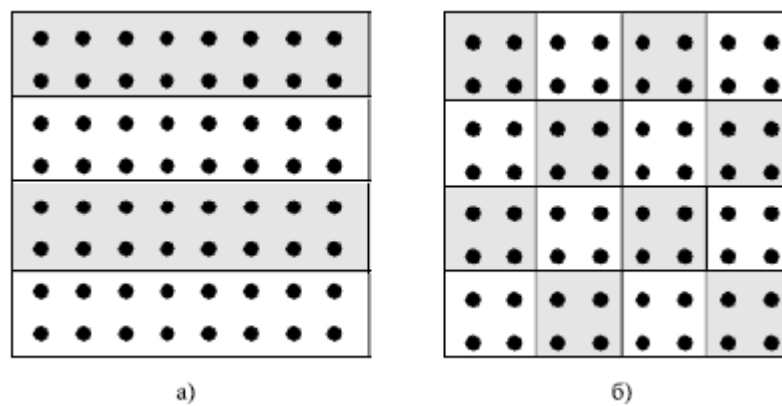


Рис. 4.3 Разделение данных на области: а – ленточная схема, б – блочная схема

Например, в случае двумерной задачи наиболее часто используется один из двух типов декомпозиции: область может быть разделена на отдельные строки (или последовательные группы строк) - так называемая ленточная схема разделения данных, либо на прямоугольные наборы элементов - блочная схема разделения данных (см. рис. 4.3). Возникает естественный вопрос: какая из этих схем декомпозиции «лучше»?

Выбор одной из указанных схем декомпозиции диктуется требованием минимизации пересылок данных между процессорами. Рассмотрим эту задачу для двумерного случая. Будем полагать, что области заданы в виде прямоугольников или квадратов, ширина полос данных в окрестности границ, которыми должны обмениваться процессоры, не зависит от направления границ фрагментов, а объем передаваемых данных определяется длиной сопряженных границ фрагментов. Приведем простой пример декомпозиции двумерной области, имеющей размеры $H \times L, H \geq L$. При декомпозиции области данных на четыре подобласти (процессора) общий объем передаваемых данных при ленточном разделении данных (вдоль стороны L) пропорционален $3L$, а при блочном (на равные прямоугольники) - $H+L$.

Объем максимального межпроцессорного обмена данными между парами процессоров, обрабатывающих соседние области, составит соответственно L и

$H/2$. Нетрудно заметить одинаковый общий и максимальный межпроцессорный обмены имеют место при $H=2L$. Если $H<2L$, выгоднее блочная декомпозиция, при $H>2L$ - ленточная. Ясно, что при другом числе процессоров (подобластей) результаты могут оказаться иными.

Если оказалось, что выгоднее блочная декомпозиция, то следующий важный вопрос - выбор размеров блоков. С точки зрения минимизации отношения длины граничных областей к их площади (пропорционального отношению объема межпроцессорного обмена к объему вычислений в данной подобласти) представляется, что форму подобластей следует взять в виде квадратов или прямоугольников близким к квадратам. Однако при этом возникает еще одна проблема.

При разбиении исходной области обработки данных на квадраты одинаковых размеров для фрагментов, расположенных на границах декомпозируемой области, длина границ, сопряженных с соседними фрагментами, а, следовательно, и объем передаваемых данных, будет меньше. Указанное различие во времени передачи данных может оказывать существенное влияние на эффективность использования процессоров, если скорость передачи данных низкая. Неэффективность использования процессоров более заметна, когда число областей, на которые разбивается изображение, невелико.

Повышение эффективности использования процессоров может быть достигнуто увеличением размеров областей, находящихся на границах и в углах изображения. В следующем разделе этот вопрос будет детально рассмотрен для случая блочной декомпозиции.

4.5. Блочная декомпозиция с учетом локализации подобластей

Известно, что весьма широкий класс задач реализуется в классе алгоритмов с распределением данных (Data Partitioning), в которых пространство данных может быть разделено на непересекающиеся области, а

вычисления могут осуществляться независимо и требуется лишь редкий обмен между этими процессами. В частности, при моделировании на основе метода конечных элементов, секционной свертке изображений и др. после каждой итерации осуществляется обмен данными, полученными на границах соседних областей. Ясно, что в случае, когда размеры областей, на которые разбивается вся область значений, одинаковы, объемы пересылаемых данных будут различаться в зависимости от места расположения области.

Решим задачу такого разбиения исходной области на квадратные блоки с учетом локализации подобластей, при котором время работы всех процессоров с учетом пересылок максимально сбалансировано. Для простоты рассмотрим случай, когда исходная область квадратная: $X \times X$, где X - число отсчетов одной стороны области. Будем полагать, что для заданных: вычислительного алгоритма и вычислительной системы, известны константы: τ_n - время расчета при обработке одного отсчета (точки) области и τ_p - среднее время, затрачиваемое на передачу информации, необходимой для одной точки области.

Обычно, с точки зрения удобства организации вычислений, всю область данных разбивают на прямоугольные фрагменты. Пусть x - сторона фрагмента, численно равная числу точек. Потребуем, чтобы величина x удовлетворяла неравенству

$$\Delta_{\text{доп}} \leq (4 \cdot x \cdot \tau) / (x^2 \tau_p) \leq (4 \cdot \delta) / x \quad (4.1)$$

где $\delta = \tau_n / \tau_p$ - отношение отрезков времени, необходимых для пересылки данных к времени обработки в расчете на одну точку области, а $\Delta_{\text{доп}}$ - допустимая величина отношения времени пересылок к времени обработки внутренней области, задаваемая из условия эффективной загрузки процессоров.

Ясно, что неравенство (4.1) выполняется также для областей, расположенных на границах исходной области, т.к. они имеют меньшую длину

сопряженных границ. Области, находящиеся в углах изображения размером $x_0 \times x_0$, будем называть угловыми, области $x_0 \times x$ ($x \times x_0$)- граничными, а области $x \times x$ - внутренними. Задача заключается в том, чтобы найти x_0 и x такие, чтобы время обработки всех областей с учетом затрат на пересылку было одинаковым (рис. 4.4).

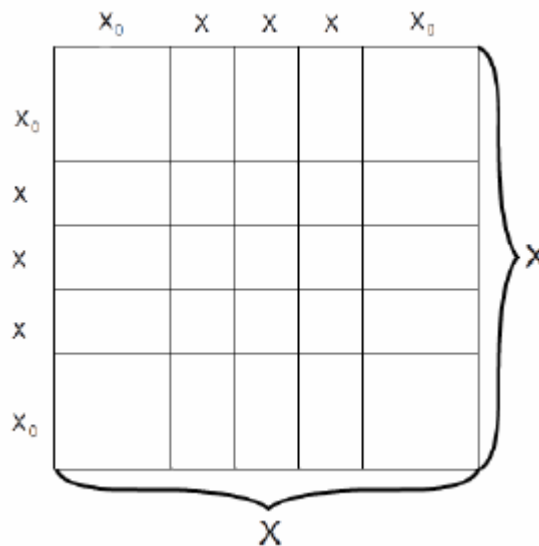


Рис. 4.4 Разбиение квадратного изображения на фрагменты

Для простоты полагаем, что ширина полос данных на границах областей, которыми они должны обмениваться, равна одному отсчету, поэтому объем передаваемых данных пропорционален длине границ. Тогда суммарное время обработки с учетом пересылок: а) для внутренней области:

а) для внутренней области:

$$T_{\text{вн}} = x^2 \cdot \tau_p + 4 \cdot x \cdot \tau_n, \quad (4.2)$$

б) для граничной:

$$T_{\text{гр}} = x \cdot x_0 \cdot \tau_p + 2 \cdot x_0 \cdot \tau_n + x \cdot \tau_n, \quad (4.3)$$

в) для угловой:

$$T_{\text{угл}} = x_0^2 \cdot \tau_p + 2 \cdot x_0 \cdot \tau_n. \quad (4.4)$$

Положим

$$x_0 = k \cdot x, \quad (4.5)$$

где $1 < k < 1,5$ - коэффициент увеличения угловой (и соответственно граничной) области, который необходимо выбрать из условия балансировки процессоров.

При балансировке процессоров, обрабатывающих внутренние и граничные области, вычислительные затраты на обработку угловых областей существенно возрастают. Поэтому потребуем, чтобы выполнялось равенство

$$T_{\text{угл}} = T_{\text{вн}},$$

или

$$k^2 \cdot x + 2 \cdot k \cdot \delta = x + 4 \cdot \delta. \quad (4.6)$$

Отбрасывая из решений (4.6) отрицательные значения k , получаем

$$k = \frac{\delta}{x} + \sqrt{\frac{\delta}{x} + 1 + \frac{4 \cdot \delta}{x}}, \quad (4.7)$$

С учетом неравенства (4.1) в соответствии с (4.7) можно записать условие для допустимых значений k :

$$k \leq -\frac{\Delta_{\text{доп}}}{x} + \sqrt{\left(\frac{\Delta_{\text{доп}}}{x}\right)^2 + 1 + 4 \cdot \Delta_{\text{доп}}}, \quad (4.8)$$

Остается подобрать удовлетворяющее условию (4.8) наибольшее значение k , при котором целое число n (число полос, на которые разбивается область решений) удовлетворяет равенству

$$(n-2)x + 2kx = X. \quad (4.9)$$

Заметим, что обычно отношение δ/x невелико, при этом для значений k , удовлетворяющих (4.8), время обработки граничных областей не превышает времени обработки угловых и внутренних областей.

Соотношения (4.8), (4.9) могут использоваться для выбора начального разбиения исходной области на фрагменты. В действительности эффективность загрузки процессоров будет зависеть от многих других факторов, которые не учитывались в нашей упрощенной модели (например, латентность при передаче данных, а также тот факт, что исходная область может быть не квадратной, а X не обязано делиться без остатка на величину x , и др.).

Для более полного учета влияния всех факторов, которые не принимались во внимание в указанной упрощенной постановке, может использоваться технология итерационного планирования распределения ресурсов, описанная в работе [9]. В данном случае ее применение не вызовет дополнительных усложнений по сравнению с описанным в указанной работе вариантом, поскольку задача выбора k однопараметрическая.

4.6. Общие рекомендации по разработке параллельных программ

Ранее мы уже подчеркивали, что при переносе последовательной программы на параллельную ЭВМ без ее существенной переработки, как правило, не приводит к ускорению вычислений. Усилия, затрачиваемые на эту переработку, в значительной степени зависят от типа решаемой задачи, а именно: допускает ли задача распараллеливание по данным (параллелизм данных) или имеет место лишь параллелизм задач [7]. Как уже отмечалось выше, переработка последовательной программы существенно облегчается, если задача допускает распараллеливание по данным. В этом случае задача переработки может свестись к разбиению массива исходных данных на фрагменты, обработка которых ведется независимо на различных процессорах. Ясно, что при этом должна обеспечиваться равномерная загрузка процессоров, с учетом их, возможно, различной производительности.

Эффективность программы будет зависеть от соотношения временных затрат на проведение вычислений на фрагментах исходных данных и пересылку данных (накладные расходы) [7]. По мере увеличения числа (а значит

уменьшения размеров) фрагментов данных объем вычислений на каждом фрагменте уменьшается. При этом накладные расходы могут оставаться почти прежними, например, вследствие большой латентности (связанной с потерями на передачу сообщения нулевой длины) коммуникационной среды.

Можно рекомендовать следующий простой способ построения эффективной программы, основанной на свойстве параллелизма данных. Размеры фрагментов массива исходных данных следует уменьшать (соответственно увеличивать число параллельно работающих процессоров) до тех пор, пока имеет место почти линейное ускорение. Если же при очередном увеличении числа процессоров линейного ускорения не происходит, это означает, что накладные расходы стали заметными и дальнейшее распараллеливание по данным приведет к недостаточной загрузке процессоров.

Если задача не допускает распараллеливания по данным, т.е. возможен лишь параллелизм задач, трудности существенно возрастают. Подход к программированию, основанный на параллелизме задач, подразумевает, что вычислительная задача разбивается на несколько относительно самостоятельных подзадач и каждый процессор загружается своей собственной подзадачей. Для каждой подзадачи пишется своя собственная программа. Чем больше подзадач, тем большее число процессоров можно использовать и тем большего ускорения можно ожидать (если удастся обеспечить равномерную загрузку процессоров и минимизировать обмен данными между ними).

Для построения эффективного кода в данном случае программист должен провести анализ затрачиваемого времени разными частями программы с целью выявления наиболее ресурсопотребляющих частей.

5. ИСПОЛЬЗОВАНИЕ МНОГОПОТОЧНОСТИ В ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЯХ

5.1. Как работает Threading

Многопоточность в .Net управляется планировщиком потоков, функцией, которую CLR обычно передает в операционную систему. Планировщик потоков гарантирует, что всем активным потокам выделяется соответствующее время выполнения, а потоки, которые ждут или заблокированы не потребляют процессорное время. На однопроцессорном компьютере планировщик потоков выполняет **разделение времени** - быстрое выполнение переключения между каждым из активных потоков. На многопроцессорном компьютере многопоточность реализуется со смесью разделения времени и подлинного параллелизма, где разные потоки одновременно запускают код на разных процессорах. Поток называется **вытеснен**, когда его выполнение прерывается внешними факторами, таких как квантование времени. В большинстве ситуаций сам поток не контролирует, когда и где он будет выгружен. Поток аналогичен процессу операционной системы, в котором выполняется приложение. Подобно тому, как процессы запускаются параллельно на компьютере, потоки выполняются параллельно в рамках одного процесса. Процессы полностью изолированы друг от друга; потоки имеют ограниченную степень изоляции. В частности, потоки разделяют (кучу) память с другими потоками, запущенными в одном приложении.

Наиболее распространение применения многопоточности это:

1. **Поддержка адаптивного пользовательского интерфейса.** Выполняя трудоемкие задачи в параллельном потоке основной поток пользовательского интерфейса может продолжать обработку событий клавиатуры и мыши.
2. **Эффективное использование заблокированного CPU.** Многопоточность полезна, когда поток ожидает ответа от другого компьютера или части оборудования. В то время как один поток блокируется во время выполнения задачи, другие потоки могут воспользоваться другим неуправляемым компьютером.

3. **Параллельное программирование.** Код, который выполняет интенсивные вычисления, может быстрее выполняться на многоядерных или многопроцессорных компьютерах, если рабочая нагрузка распределяется между несколькими потоками.
4. **Одновременная обработка запросов.** На сервере клиентские запросы могут поступать одновременно, поэтому их нужно обрабатывать параллельно.

Многопоточность также несет затраты ресурсов и ЦП в планировании и переключении потоков. Многопоточность не всегда ускоряет приложение - оно может даже замедлить работу при неправильном использовании.

5.2. Создание и запуск потоков. Передача данных в поток.

Потоки создаются с использованием конструктора класса **Thread**, передавая в **ThreadStart** делегат, который указывает, где должно начаться выполнение. Вот как определяется делегат **ThreadStart**:

```
public delegate void ThreadStart ();
```

Выполнение потока продолжается до тех пор, пока не выполнится его метод, после чего поток заканчивается. Ниже приведен пример использования расширенного синтаксиса C# для создания **ThreadStart** делегата:

```
class ThreadTest  
{  
    static void Main()  
    {  
        Thread thread1 = new Thread(new ThreadStart(Go));  
        thread1.Start();  
        Go();  
    }  
    static void Go()  
    {  
        Console.WriteLine("hello!");
```



```
}  
}
```

В этом примере поток **thread1** выполняет выполняет метод **Go()** и то же время, этот метод выполняется в основном потоке. В результате получается два близких к мгновенному вывода **hello**.

Самый простой способ передать аргументы целевому методу потока - выполнить лямбда-выражение, которое вызывает метод с нужными аргументами:

class Program

```
{  
    static void Main()  
    {  
        Thread thread1 = new Thread(() => Print("Hello from thread1!"));  
        thread1.Start();  
    }  
    static void Print(string message)  
    {  
        Console.WriteLine(message);  
    }  
}
```

Другой метод – это передать аргумент в метод **ThreadStart**:

class Program

```
{  
    static void Main()  
    {  
        Thread thread1 = new Thread(Print);  
        thread1.Start("Hello from thread1!");  
    }  
    static void Print(object messageObj)
```

```

{
    string message = (string)messageObj;
    Console.WriteLine(message);
}
}

```

Это работает, потому что конструктор **Thread** перегружен, чтобы принять любого из двух делегатов:

```

public delegate void ThreadStart ();
public delegate void ParameterizedThreadStart ( object obj );

```

Ограничение **ParameterizedThreadStart** состоит в том, что он принимает только один аргумент.

5.3. Основные свойства потоков

Каждый поток имеет свойство **Name**, которое можно использовать для отладки. Имя потока можно установить только один раз, попытки изменить его позже вызывают исключение. Статическое свойство **Thread.CurrentThread** предоставляет собой исполняемый поток. В следующем примере мы задаем имя основного потока:

```

class ThreadNaming
{
    static void Main()
    {
        Thread.CurrentThread.Name = "main";
        Thread workerThread = new Thread(Go);
        workerThread.Name = "worker";
        workerThread.Start();
        Go();
    }

    static void Go()

```

```

{
    Console.WriteLine("Hello from " + Thread.CurrentThread.Name);
}
}

```

По умолчанию потоки, созданные вами явно, являются **потоками переднего плана**. Потоки переднего плана поддерживают приложение до тех пор, пока работает любой из них. Как только все передние потоки завершатся, приложение заканчивается, и любые фоновые потоки, которые все еще работают будут завершены. Статус потока переднего плана или фонового потока не имеет никакого отношения к его приоритету или распределению времени выполнения. Изменить статус потока можно используя свойство **IsBackground**.

Свойство **Priority** потока определяет, сколько времени выполнения он получает относительно других активных потоков в операционной системе, которое задается через следующее перечисление:

enum ThreadPriority{ Lowest, BelowNormal, Normal, AboveNormal, Highest }.

Это свойство становится актуальным только тогда, когда одновременно задействованы несколько потоков.

5.4. Синхронизация выполнения потоков

Синхронизация - это координация действий потоков для получения прогнозируемого результата. Синхронизация особенно важна, когда потоки обращаются к одним и тем же данным.

Конструкции синхронизации можно разделить на четыре категории:

1. **Простые методы блокировки.** Они ждут окончания другого потока или в течение определенного периода времени. Методы **Sleep, Join** являются простыми методами блокировки.
2. **Блокирующие конструкции.** Они ограничивают количество потоков, которые могут выполнять некоторую активность или выполнять секцию кода за раз. Эксклюзивные блокирующие конструкции наиболее

распространены - они позволяют только один поток за раз и позволяют конкурирующим потокам получать доступ к общим данным, не мешая друг другу. Стандартными эксклюзивными запирающими конструкциями являются **lock**, **Mutex**, **SpinLock**. Неэксклюзивные блокирующие конструкции - это **Semaphore**, **SemaphoreSlim**.

3. **Сигнализационные конструкции.** Они позволяют потоку приостанавливаться до получения уведомления от другого, избегая необходимости в неэффективном опросе. Наиболее часто используемые сигнальные устройства - это дескрипторы ожидания событий и класс **Monitor**.

4. **Неблокирующие конструкции синхронизации.** Они защищают доступ к общему полю, вызывая примитивы процессора. В CLR и C# предоставляют следующие неблокирующие конструкции: **Thread.MemoryBarrier**, **Thread.VolatileRead**, **Thread.VolatileWrite**, по **volatile** ключевому слову, и класс **Interlocked**.

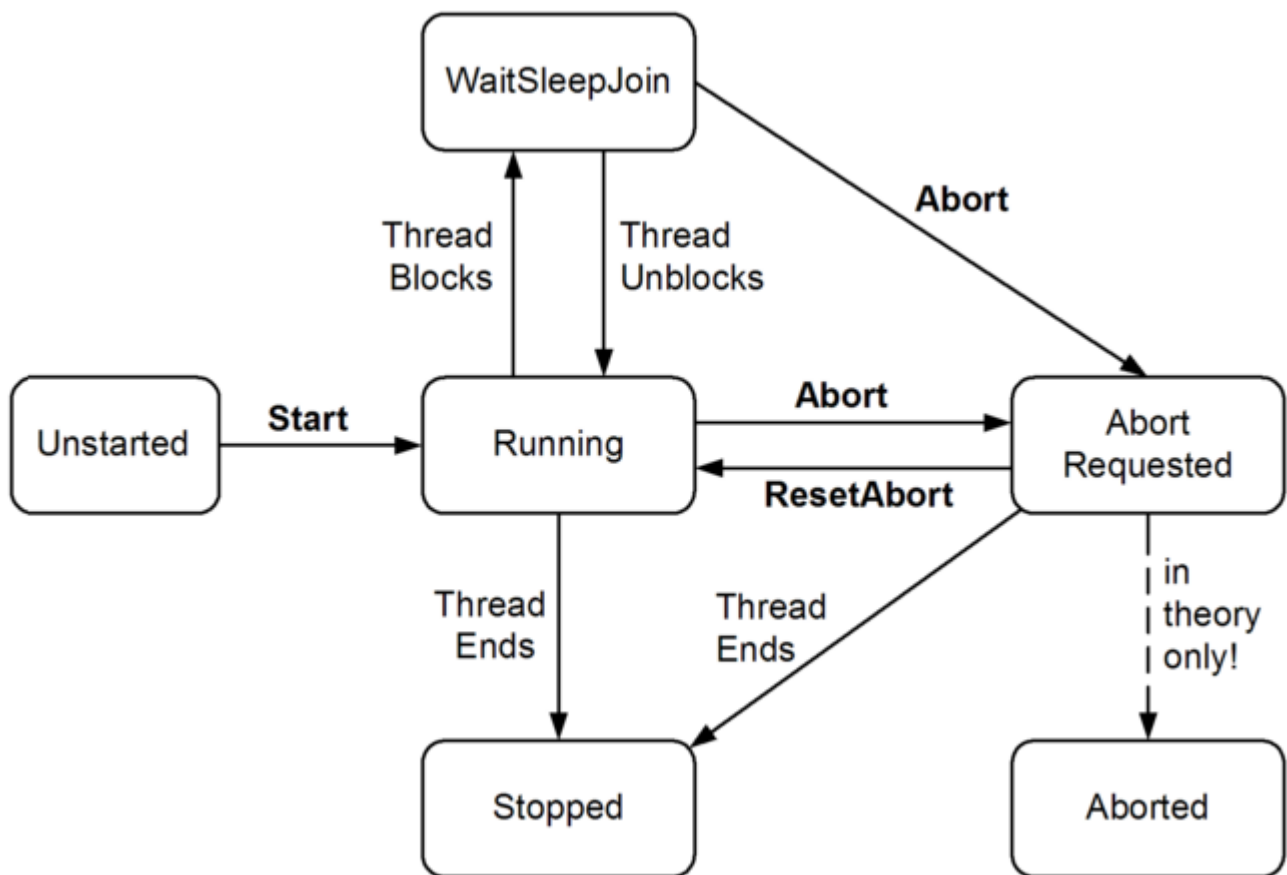
Блокировка важна для всех, кроме последней категории. Давайте кратко рассмотрим это понятие. Поток считается **заблокированным**, когда его выполнение приостанавливается по какой-либо причине, например, при вызове метода **Sleep** или ожидании завершения другого через вызов методов **Join** или **EndInvoke**. Блокированный поток немедленно отдает свое процессорное время, и с этого момента не потребляет процессорное время, пока не будет выполнено его условие блокировки. Проверить, заблокирован ли поток можно через его свойство **ThreadState**. Когда поток блокируется или разблокируется, операционная система переключает контекст. Разблокировка происходит одним из четырех способов:

1. по условию блокировки
2. по тайм-ауту операции (если задан тайм-аут)
3. путем прерывания через **Thread.Interrupt**
4. путем прерывания через **Thread.Abort**

Поток не считается заблокированным, если его выполнение приостанавливается с помощью устаревшего метода **Suspend**.

5.5. Статус выполнения потока

Статус выполнения потока можно узнать через его свойство **ThreadState**. Это свойство возвращает элемент перечисления типа **ThreadState**. Большинство значений, однако, являются избыточными, неиспользуемыми или устаревшими. На следующей диаграмме показаны состояния потоков и способы изменения состояния:



Свойство **ThreadState** полезно для диагностических целей, но непригодно для синхронизации, потому что состояние потока может измениться между тестированием **ThreadState** и действует на основе этой информации.

5.6. Блокировка

Исключительная блокировка используется для обеспечения того, чтобы только один поток мог вводить определенные разделы кода за раз. Две

основные эксклюзивные блокировки - это оператор **lock** и **Mutex**. Оператор **lock** быстрее и удобнее. Рассмотрим следующий пример:

```
class ThreadUnsafe
```

```
{  
    static int _val1 = 1, _val2 = 1;  
    static void Go()  
    {  
        if (_val2 != 0)  
            Console.WriteLine(_val1 / _val2);  
        _val2 = 0;  
    }  
}
```

Этот класс не является потокобезопасным, если метод **Go** будет вызван двумя потоками одновременно, то можно было бы получить ошибку деления на нуль, потому что **_val2** можно было бы установить нуль в одном потоке прямо, так как другой поток находился между выполнением **if** оператора и **Console.WriteLine()**.

Вот как с помощью оператора **lock** можно решить проблему:

```
class ThreadSafe
```

```
{  
    static readonly object _locker = new object();  
    static int _val1, _val2;  
    static void Go()  
    {  
        lock (_locker)  
        {  
            if (_val2 != 0) Console.WriteLine(_val1 / _val2);  
            _val2 = 0;  
        }  
    }
```

```
}  
}
```

Только один поток может блокировать синхронизирующий объект (в данном случае **_locker**) за раз, и любые конфликтующие потоки блокируются до тех пор, пока блокировка не будет освобождена. Если более чем один поток ссылается на блокировку, они ставятся в очередь и предоставляют блокировку на основе «первым пришел, первым обслужен». В этом случае мы защищаем логику внутри метода **Go**, а также поля **_val1** и **_val2**.

Оператор lock C # на самом деле является синтаксическим ярлыком для вызова методов **Monitor.Enter** и **Monitor.Exit** с помощью блока **try/finally**. Вот что на самом деле происходит в **Go** методе предыдущего примера:

```
Monitor.Enter (_locker);  
  
try  
{  
    if (_val2 != 0) Console.WriteLine (_val1 / _val2);  
    _val2 = 0;  
}  
finally { Monitor.Exit (_locker);  
}
```

Любой объект, видимый для каждого из сторонних потоков, может использоваться как объект синхронизации, подчиненный одному жесткому правилу: он должен быть ссылочным типом. Синхронизирующий объект обычно является приватным (поскольку это помогает инкапсулировать логику блокировки) и обычно является экземпляром или статическим полем.

Недостатком блокировки таким образом является то, что вы не инкапсулируете логику блокировки, поэтому становится труднее предотвратить чрезмерную блокировку. Блокировка может также

просачиваться через границы домена приложения (в рамках одного и того же процесса).

Вы также можете блокировать локальные переменные, захваченные лямбда-выражениями или анонимными методами. Блокировка не ограничивает доступ к самому синхронизирующему объекту. Другими словами, **x.ToString()** он не будет блокироваться, потому что вызвал другой поток **lock(x)**; оба потока должны вызывать **lock(x)** для блокировки.

В качестве основного правила вам необходимо заблокировать доступ к любому доступному для записи области. Даже в простейшем случае - операция присваивания в одном поле - вы должны учитывать синхронизацию. В следующем классе ни метод, **Increment** ни **Assign** метод не являются потокобезопасными:

```
class ThreadUnsafe { static int _x ; static void Increment () { _x ++; } static void Assign () { _x = 123 ; } }
```

Вот потокобезопасные версии **Increment** и **Assign**:

```
class ThreadSafe { static readonly object _locker = new object (); static int _x ;  
    static void Increment () { lock ( _locker ) _x ++; }  
    static void Assign () { lock ( _locker ) _x = 123 ; }  
}
```

Если группа переменных всегда читается и записывается в пределах одной блокировки, вы можете сказать, что переменные читаются и записываются атомарно. Предположим, что поля **x** и **y** всегда читаются и назначаются внутри **lock** объекта **locker**:

```
lock ( locker ) { if ( x != 0 ) y /= x ; }
```

Можно сказать что к переменным **x** и **y** мы получаем доступ атомарно, потому что кодовый блок не может быть разделен или вытеснен действиями другого потока таким образом, чтобы он изменил **x** или **y** до вычисления результата.

Атомность, предоставляемая блокировкой, нарушается, если выбрасывается исключение внутри **lock** блока. Например, рассмотрим следующий пример:

```
decimal _savingsBalance, _checkBalance;
void Transfer (decimal amount)
{
    lock (_locker)
    {
        _savingsBalance += amount;
        _checkBalance -= amount + GetBankFee();
    }
}
```

Если бы исключение было брошено методом **GetBankFee()**, банк потерял бы деньги. В этом случае мы могли бы избежать проблемы, вызвав **GetBankFee** ранее. Решением для более сложных случаев является реализация логики «отката» внутри **catch** или **finally** блока.

Инструкция является атомарной, если она выполняет нераздельно на основном процессоре.

5.7. Mutex

Еще один инструмент управления синхронизацией потоков представляет класс **Mutex**, также находящийся в пространстве имен **System.Threading**. Данный класс является классом-оболочкой над соответствующим объектом ОС Windows "мьютекс". Мьютекс представляет собой взаимно исключающий синхронизирующий объект. Это означает, что он может быть получен потоком только по очереди. Мьютекс предназначен для тех ситуаций, в которых общий ресурс может быть одновременно использован только в одном потоке. Допустим, что системный журнал совместно используется в нескольких процессах, но только в одном из них данные могут записываться в файл этого журнала в любой момент времени. Для синхронизации процессов в данной

ситуации идеально подходит мьютекс. Приобретение и освобождение **Mutex** занимает несколько микросекунд - примерно в 50 раз медленнее чем **lock**.

В конструкторе класса **Mutex** указывается, должен ли мьютексом изначально владеть вызывающий поток, и его имя. У **Mutex** имеется несколько конструкторов. Ниже приведены два наиболее употребительных конструктора:

```
public Mutex()
```

```
public Mutex(bool initiallyOwned)
```

В первой форме конструктора создается мьютекс, которым первоначально никто не владеет. А во второй форме исходным состоянием мьютекса завладевает вызывающий поток, если параметр **initiallyOwned** имеет логическое значение **true**. В противном случае мьютексом никто не владеет.

Для того чтобы получить мьютекс, в коде программы следует вызвать метод **WaitOne()** для этого мьютекса. Метод **WaitOne()** наследуется классом **Mutex** от класса **Thread.WaitHandle**. Метод **WaitOne()** ожидает до тех пор, пока не будет получен мьютекс потока из которого он был вызван. Следовательно, этот метод блокирует выполнение вызывающего потока до тех пор, пока не станет доступным указанный мьютекс. Приведем пример создания и использования мьютекса:

```
class OneAtATimePlease
```

```
{
```

```
    static void Main()
```

```
{
```

```
    using (var mutex = new Mutex(false, "Mutex1"))
```

```
{
```

```
        if (!mutex.WaitOne(TimeSpan.FromSeconds(3), false))
```

```
        {
```

```
            Console.WriteLine("Another app instance is running. Bye!");
```

```
            return;
```

```

    }
    RunProgram();
}
Console.ReadLine();
}

static void RunProgram()
{
    Console.WriteLine("Running. Press Enter to exit");
    Console.ReadLine();
}
}

```

5.8. Семафор

Еще один инструмент, который предлагает нам платформа .NET для управления синхронизацией, представляют семафоры. Семафоры позволяют ограничить доступ определенным количеством объектов. Семафор аналогичен **Mutex** за исключением того, что семафор не имеет «владельца». Любой поток может вызывать метод **Release** семафора, тогда как с **Mutex** и **lock**, только поток, который получил блокировку, может ее освободить. Семафоры могут быть полезны в ограничении параллелизма - предотвращения слишком большого количества потоков от выполнения определенной части кода одновременно. В следующем примере пять потоков пытаются ввести ночной клуб, который позволяет одновременно задействовать только три потока:

```

class TheClub
{
    static SemaphoreSlim _sem = new SemaphoreSlim(3,3);
    static void Main()
    {
        for (int i = 1; i <= 5; i++) new Thread(Enter).Start(i);
    }
}

```

```

    }

    static void Enter(object id)
    {
        Console.WriteLine(id + " wants to enter");
        _sem.Wait();
        Console.WriteLine(id + " is in!");
        Thread.Sleep(1000 * (int)id);
        Console.WriteLine(id + " is leaving");
        _sem.Release();
    }
}

```

Для создания семафора используется конструктор класса **Semaphore**: *static Semaphore sem = new Semaphore(3, 3)*. Его конструктор принимает два параметра: первый указывает, какому числу объектов изначально будет доступен семафор, а второй параметр указывает, какой максимальное число объектов будет использовать данный семафор. В данном случае у нас только три потока могут одновременно находиться в ночном клубе, поэтому максимальное число равно 3. Основной функционал сосредоточен в методе **Enter**, который и выполняется в потоке. В начале для ожидания получения семафора используется метод **Wait**. После того, как в семафоре освободится место, данный поток заполняет свободное место и начинает выполнять все дальнейшие действия. После окончания чтения мы высвобождаем семафор с помощью метода **Release**. После этого в семафоре освобождается одно место, которое занимает другой поток.

5.9. Сигнализация с помощью классов **EventWaitHandle**

Обработчики событий **EventWaitHandle** используются для **сигнализации**. Сигнализация - это когда один поток ждет, пока он не получит уведомление от другого. Обработчики событий - это самые простые из

сигнальных конструкций, и они не связаны с событиями C#. Они бывают трех видов: **AutoResetEvent**, **ManualResetEvent**. Они основаны на общем **EventWaitHandle** классе, от которого они получают всю свою функциональность. **AutoResetEvent** похож на турникет с билетами: вставка билета позволяет пройти ровно одному человеку. «**Auto**» в названии класса относится к тому факту, что открытый турникет автоматически закрывается или «сбрасывается» после того, как кто-то через него пройдет. Вы можете создать объект **AutoResetEvent** двумя способами. Первый – через вызов его конструктора:

```
var auto = new AutoResetEvent ( false );
```

Второй способ создания **AutoResetEvent** :

```
var auto = new EventWaitHandle (false, EventResetMode.AutoReset);
```

Применяются события очень просто. Для события типа **ManualResetEvent** порядок применения следующий. Поток, ожидающий некоторое событие, вызывает метод **WaitOne()** для событийного объекта, представляющего данное событие. Если событийный объект находится в сигнальном состоянии, то происходит немедленный возврат из метода **WaitOne()**. В противном случае выполнение вызывающего потока приостанавливается до тех пор, пока не будет получено уведомление о событии. Как только событие произойдет в другом потоке, этот поток установит событийный объект в сигнальное состояние, вызвав метод **Set()**. После установки событийного объекта в сигнальное состояние произойдет немедленный возврат из метода **WaitOne()**, и первый поток возобновит свое выполнение. А в результате вызова метода **Reset()** событийный объект возвращается в несигнальное состояние.

Событие **AutoResetEvent** отличается от события типа **ManualResetEvent** лишь способом установки в исходное состояние. Если для события типа **ManualResetEvent** событийный объект остается в сигнальном состоянии до тех пор, пока не будет вызван метод **Reset()**, то для события типа **AutoResetEvent** событийный объект автоматически переходит в несигнальное состояние, как

только поток, ожидающий это событие, получит уведомление о нем и возобновит свое выполнение. Поэтому если применяется событие типа **AutoResetEvent**, то вызывать метод **Reset()** необязательно.

В следующем примере начинается поток, который просто ждет пока не будет сигнализирован другим потоком:

```
class BasicWaitHandle
{
    static EventWaitHandle _waitHandle = new AutoResetEvent(false);

    static void Main()
    {
        new Thread(Waiter).Start();
        Thread.Sleep(1000);
        _waitHandle.Set();
    }

    static void Waiter()
    {
        Console.WriteLine("Waiting...");
        _waitHandle.WaitOne();
        Console.WriteLine("Notified");
    }
}
```

6. ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ В .NET 4.0

6.1. Введение

В этом разделе мы рассмотрим классы параллельного программирования, предоставляемые платформой .NET. Мы рассмотрим две библиотеки. Это

параллельная библиотека задач (TPL) и параллельная версия языкового интегрированного запроса (PLINQ).

Параллельная библиотека задач (TPL) обеспечивает параллелизм, основанный на декомпозиции данных и задач. Параллелизм данных упрощается с помощью новых версий циклов `for` и `foreach`, которые автоматически декомпилируют данные и разделяют итерации на все доступные процессорные ядра. Параллелизм задач обеспечивается новыми классами, которые позволяют определять задачи с помощью *лямбда-выражений*. Вы можете создавать задачи, а платформа .NET будет определять, когда они будут выполняться, и какие из доступных процессоров будут выполнять эту работу. TPL обеспечивает императивную форму параллельного программирования.

Параллельный LINQ является декларативным, а не императивным, как и последовательная версия LINQ. Такой подход к параллелизму имеет более высокий уровень, обеспечиваемый TPL. Он позволяет использовать стандартные операторы запросов и автоматически назначать работу, выполняемую одновременно доступными процессорами.

Новые функции параллельного программирования в платформе .NET обеспечивают несколько преимуществ, которые делают его предпочтительным по сравнению со стандартной многопоточностью. При ручном создании потоков вы можете создавать слишком много потоков, что приводит к чрезмерным операциям переключения задач, которые влияют на производительность. TPL PLINQ обеспечивают автоматическую декомпозицию данных. Важно понимать, что новые библиотеки обеспечивают **потенциальный** параллелизм. При стандартной многопоточности при запуске нового потока он сразу начинает свою работу. Это может быть не самый эффективный способ использования доступных процессоров. Библиотеки параллелизма могут запускать новые потоки, если процессорные ядра доступны. Если это не так, задачи могут быть

отложены до тех пор, пока ядро не станет свободным или пока результат операции не понадобится.

Наконец, новые библиотеки позволяют не беспокоиться о количестве доступных ядер и количестве, которое может быть доступно. Все необходимые ядра будут использоваться по мере необходимости. Если код выполняется на однопроцессорной машине, он будет в основном выполняться последовательно. параллельный код, работающий на одноядерном компьютере, будет работать медленнее, чем просто последовательный код из-за накладных расходов, связанных с библиотеками параллелизма. Однако это влияние незначительно по сравнению с полученными преимуществами.

6.2. Параллельный цикл **For**

Параллельная библиотека задач (TPL) включает в себя две команды цикла, которые являются параллельными версиями операторов цикла **for** и **foreach** для C#. Каждый из них предоставляет код, необходимый для шаблона параллельного цикла, гарантируя, что весь процесс будет завершен с выполнением всех итераций, прежде чем перейти к следующему циклу. Отдельные итерации разбиваются на группы, которые могут быть разделены между доступными процессорами, что повышает производительность на машинах с несколькими ядрами.

Parallel.For позволяет создать цикл с фиксированным числом итераций. Если доступно несколько ядер, итерации можно разложить на группы, которые выполняются параллельно. Чтобы продемонстрировать, создадим новое консольное приложение. Параллельные циклы находятся в пространстве имен **System.Threading.Tasks**, поэтому необходимо добавить в начало программы:

using System.Threading.Tasks;

Синтаксис параллельного цикла отличается тем что он предоставляется статическим методом, а не ключевым словом C#. Версия интересующего метода имеет три параметра. Первые два аргумента определяют

нижнюю и верхнюю границы цикла, причем верхняя граница является исключительной. Третий параметр принимает делегат, обычно выражаемый через лямбда выражение, который содержит код для запуска во время каждой итерации.

```
Parallel.For(0, 10, i =>  
{  
    long total = GetTotal();  
    Console.WriteLine("{0} - {1}", i, total);  
});
```

Если вы запустите приведенный выше код на компьютере с несколькими ядрами вы должны увидеть значительное улучшение производительности. В одноплатформенной одноплатформенной системе производительность будет незначительно медленнее, чем эквивалентный последовательный цикл.

Существуют различные подводные камни с которыми можно столкнуться при использовании параллельных циклов. Некоторые из них сразу же приводят к очевидным ошибкам в коде. Некоторые из них вызывают тонкие ошибки, которые могут возникать редко и трудно найти. Другие просто снижают производительность параллельных циклов.

6.3. Параллельный цикл **ForEach**

Параллельный цикл **ForEach** обеспечивает параллельную версию стандартного последовательного цикла **foreach**. Каждая итерация обрабатывает один элемент из коллекции. Однако параллельный характер цикла означает, что несколько итераций могут выполняться одновременно на разных процессорах или процессорных ядрах.

Параллельная версия цикла использует статический метод **ForEach** класса **Parallel**. Существует много перегруженных версий этого метода. Наиболее простой принимает два аргумента. Первый - это коллекция объектов, которые будут перечислены. Это может быть любая коллекция, которая реализует **IEnumerable <T>**.

Второй параметр - делегат, обычно выражаемый как лямбда-выражение, который определяет действие, которое необходимо предпринять для каждого элемента в коллекции. Параметр делегата содержит элемент из коллекции, который должен обрабатываться во время итерации.

Обратите внимание, что тело цикла теперь является телом оператора `lambda`, и коллекция передается первому параметру метода.

```
Parallel.ForEach(Enumerable.Range(1, 10), i =>  
{  
    Console.WriteLine("{0} - {1}", i, GetTotal());  
});
```

6.4. Завершение параллельных циклов

Все стандартные циклы, предоставляемые *C#*, а именно циклы `for`, `foreach` и `while` дают возможность досрочного выхода из цикла используя команду **break**. Когда встречается этот оператор, цикл останавливается немедленно, любые оставшиеся итерации отменяются, и программа продолжается с команды, следующей за циклом. Это полезно в тех случаях, когда неэффективно продолжать цикл. Например, если вы перебираете набор значений для поиска определенного элемента, вы можете выйти из цикла, когда элемент найден.

Когда вы работаете с параллельными циклами, выходить не так просто, как с последовательными циклами. Поскольку несколько итераций цикла могут выполняться параллельно, выход из одной итерации на одном процессоре должен быть синхронизирован с другими итерациями, выполняемыми на других ядрах. Если параллельные операции были остановлены, возможно, что данные будут оставлены в несогласованном состоянии, так что другие итерации будут продолжать работать. Первой проблемой, с которой вы можете столкнуться, является то, что команда **break**, которая используется с последовательными циклами, недоступна для их параллельных версий этого

метода. Команда **break** недоступна, поскольку параллельные циклы предоставляются статическими методами класса **Parallel**, а не как часть языка C#. Команда **break** - ключевое слово C#, которое работает только с непараллельными циклами for C#.

Чтобы согласовать итерации параллельных циклов, в том числе выходить из этих циклов по мере необходимости, мы должны использовать экземпляр класса **ParallelLoopState**. Класс **ParallelLoopState** позволяет итерациям параллельных циклов взаимодействовать друг с другом. Одним из методов класса **ParallelLoopState** является **Break**. Он похож на оператор **break** для последовательных циклов. В приведенном ниже коде показан метод в действии. Параллельный цикл запускает итерации в соответствии с количеством доступных ядер. Затем он обрабатывает каждую итерацию, проверяя, превышает ли значение счетчика цикла значение пятнадцати. Когда такое значение найдено, выполняется метод **Break**. Обратите внимание, что переменная **ParallelLoopState** с именем «pls» не создается напрямую.

```
Parallel.For(1, 20, (i, pls) =>
{
    Console.WriteLine(i + " ");
    if (i >= 15)
    {
        Console.WriteLine("Break on {0}", i);
        pls.Break();
    }
});
```

Метод **Break** пытается имитировать команду **break** последовательных циклов. В частности, он пытается обеспечить, чтобы все итерации, которые были бы выполнены последовательно, будут обрабатываться в параллельном цикле. Когда любое ядро вызывает **Break**, номер итерации записывается в объект **ParallelLoopState**. Это становится номером последней итерации,

которая может быть выполнена. Другие потоки будут продолжать работать до тех пор, пока они не достигнут этого номера итерации или не найдут другой оператор **Break**, который еще больше снизит число. Общий результат состоит в том, что все итерации, которые были бы обработаны в последовательном цикле, должны обрабатываться в параллельной версии. Однако в параллельном цикле может выполняться еще много итераций. Также возможно, что один и тот же параллельный цикл может запускать различный набор итераций при отдельных исполнениях. Вы должны учитывать эти возможности, когда вы создаете параллельный цикл, который может быть завершен досрочно.

В некоторых ситуациях вам потребуется, чтобы ваш параллельный цикл выходил как можно быстрее, не пытаясь имитировать результаты последовательного цикла. В этих случаях вы можете использовать метод **ParallelLoopState.Stop**. Как и в случае метода **Break**, итерации, выполняемые параллельно, будут завершены до того, как цикл окончательно остановится.

В следующем примере показано использование метода **Stop**. Здесь мы выполняем цикл для каждого из значений от одного до двадцати. Когда мы находим значение, которое делит на шесть без остатка, вызывается метод **Stop**.

```
Parallel.For(1, 20, (i, pls) =>
```

```
{  
    Console.WriteLine(i + " ");  
    if (i % 6 == 0)  
    {  
        Console.WriteLine("Stop on {0}", i);  
        pls.Stop();  
    }  
});
```

6.5. Исключения и параллельные циклы

Когда исключение генерируется внутри последовательного цикла то нормальный поток выполнения программы прерывается. Управление переходит к следующему доступному блоку **catch** или, если нет соответствующих инструкций **try/catch**, необработанное исключение передается в среду выполнения **.NET**, и программа прерывается. Когда блок **try/catch** присутствует, но он не находится в цикле, дальнейшие итерации не выполняются, и текущая итерация заканчивается раньше. Когда вы работаете с параллельными циклами **For** или **ForEach**, обработка исключений несколько усложняется. Когда исключение генерируется в одном потоке выполнения, вполне вероятно, что существуют итерации цикла, выполняемые параллельно. Их нельзя просто прекратить, поскольку это может привести к несоответствиям в программе. Чтобы предотвратить такие ошибки данных, итерации цикла, которые уже были запланированы для других потоков, будут продолжены.

Для обработки исключений платформа **.NET** предоставляет класс **AggregateException**. Класс **AggregateException** является подклассом класса **Exception**, и поэтому обеспечивает все стандартные функциональные возможности исключения. Кроме того, он обладает свойством, которое содержит коллекцию внутренних исключений. При вызове параллельного цикла все связанные исключения включаются в свойство, гарантируя, что данные всех исключений не будут потеряны.

Для первой демонстрации мы создадим параллельный цикл, который генерирует исключение. Поскольку мы знаем, что исключение будет обернуто в исключение **AggregateException**, мы поймем только атрибуты **AggregateExceptions**. Цикл **For** ниже итерации осуществляется через значения от -10 до 9. Каждое значение используется как делитель в простой арифметической операции. Когда значение равно нулю, возникает деление на нуль и генерируется исключение.

```

try
{
    Parallel.For(-10, 10, i =>
    {
        Console.WriteLine("100/{0}={1}", i, 100 / i);
    });
}
catch (AggregateException ex)
{
    Console.WriteLine(ex.Message);
}

```

Вывод на консоль: **Произошла одна или несколько ошибок.**

Вы можете видеть, что в какой-то момент процесса произошло деление на ноль, и после того, как цикл был остановлен, было выброшено исключение **AggregateException** и выведено его свойство **Message**. Сообщение простое, что указывает на то, что исключение **AggregateException** содержит одно или несколько внутренних исключений. Если бы мы попытались поймать исключение **DivideByZeroException**, исключение было бы необработанным.

Невозможно просмотреть приведенные выше результаты и понять, когда действительно произошло исключение. В последовательном цикле мы могли предположить, что это было во время последней обработанной итерации, но в параллельном цикле это может быть неверно. Чтобы понять, что на самом деле происходит, мы можем добавить дополнительную строку в код, который говорит нам, когда мы собираемся делить на ноль. Изменим код чтобы показать сообщение перед исключением:

```

try
{
    Parallel.For(-10, 10, i =>
    {

```

```

        if (i == 0) Console.WriteLine("About to divide by zero.");
        Console.WriteLine("100/{0}={1}", i, 100 / i);
    });
}
catch (AggregateException ex)
{
    Console.WriteLine(ex.Message);
}

```

Мы видим, что исключение произошло очень рано в процессе, после того, как были показаны только два результата расчета. Остальные вычисления произошли после первоначального исключения в ранее запланированных итерациях цикла.

После того как вы поймаете **AggregateException**, вы можете изучить каждое из содержащихся исключений, прочитав свойство **InnerExceptions**. Это свойство возвращает коллекцию только для чтения, которая может быть указана индексом или перечислены.

Пример кода ниже демонстрирует использование свойства **InnerExceptions**,

```

try
{
    Parallel.For(-10, 10, i =>
    {
        Console.WriteLine("100/{0}={1}", i, 100 / (i % 10));
    });
}
catch (AggregateException ex)
{
    foreach (Exception inner in ex.InnerExceptions)
    {
        Console.WriteLine(inner.Message);
    }
}

```

```
}  
  
}
```

6.6. Параллельность задач и Использование **Parallel.Invoke**

Некоторые алгоритмы не позволяют использовать параллелизм данных, поскольку они не повторяют одно и то же действие. Однако они могут быть кандидатами на разложение на задачи. Здесь алгоритм разбивается на части, которые могут выполняться независимо. Каждая часть рассматривается как отдельная задача, которая может выполняться на собственном ядре процессора, при этом одновременно выполняются несколько задач. Этот тип декомпозиции обычно сложнее реализовать, и иногда требуется, чтобы алгоритм был существенно изменен или полностью заменен, чтобы минимизировать элементы, которые должны выполняться последовательно и ограничивать общие изменчивые значения.

Метод **Parallel.Invoke** обеспечивает простой способ создания и выполнения нескольких задач одновременно. Как и другие методы в параллельной библиотеке задач (TPL), метод **Parallel.Invoke** обеспечивает потенциальный параллелизм. Для использования **Parallel.Invoke** выполняемые задачи предоставляются в виде делегатов. Задачи обычно определяются с помощью лямбда-выражений, но вместо них могут использоваться анонимные методы и простые делегаты. После вызова этого метода все задачи выполняются до продолжения обработки с помощью команды, следующей за инструкцией **Parallel.Invoke**. Порядок выполнения отдельных делегатов не гарантируется, поэтому вам не следует полагаться на результаты одной операции, доступной для той, которая появляется позже в массиве параметров.

Следующий пример использует **Parallel.Invoke** для запуска трех отдельных задач. Каждый из них отображает сообщение, чтобы показать, что задача запущена. Затем происходит пауза между одной-пятью секундами, пока не выдается второе сообщение, указывающее на завершение задания.


```
Parallel.Invoke(
    () => {
        Console.WriteLine("Task 1 started");
        Thread.Sleep(5000);
        Console.WriteLine("Task 1 complete");
    },
    () => {
        Console.WriteLine("Task 2 started");
        Thread.Sleep(3000);
        Console.WriteLine("Task 2 complete");
    },
    () => {
        Console.WriteLine("Task 3 started");
        Thread.Sleep(1000);
        Console.WriteLine("Task 3 complete");
    });
```

Вы можете видеть, что все три задачи выполнялись параллельно, и результаты варьируются при разных запусках программы.

Метод **Parallel.Invoke** позволяет неявно генерировать набор задач, которые могут выполняться параллельно. Когда нужно больше контролировать параллельные задачи, вы можете использовать класс **Task**. Это позволяет явно создавать параллельные задачи. Код, необходимый для создания явной задачи, несколько сложнее, чем для **Parallel.Invoke**, но преимущества перевешивают этот недостаток.

Класс **Task** может выполнять те же функции, что и **Parallel.Invoke**. Кроме того, могут быть созданы следующие типы задач:

- **Задачи продолжения.** Этот тип задачи настроен на запуск только после завершения другой задачи или группы задач. Они могут безопасно использовать данные, созданные этими более ранними задачами.

- **Вложенные и дочерние задачи.** Вложенные задачи - это просто задачи, которые создаются в рамках другой задачи, но остаются независимыми. Дочерние задачи аналогичны тем, которые созданы в родительской задаче. Однако они более тесно связаны с этим родителем.
- **Задачи с возвращаемыми значениями.** При использовании явных заданий задача может выполняться параллельно и также же возвращать значение, которое может использоваться после завершения задачи.

Класс **Task** предоставляет оболочку для делегата Action. Простым способом создания задачи является использование конструктора с единственным параметром, который принимает делегат, который вы хотите выполнить. Во многих случаях этот делегат определяется как лямбда выражение. Задачи не выполняются сразу после создания. Чтобы запустить задачу необходимо вызвать ее метод **Start()**. Следующий код имитирует шаги, которые могут быть выполнены при запуске приложения управления взаимоотношениями с клиентами (CRM). Первоначально отображается сообщение, указывающее, что приложение запускается. Затем определяются две задачи. Первая имитирует загрузку пользовательских данных из базы данных. Вторая имитирует получение данных клиента. Задачи запускаются двумя вызовами метода **Start()** до отображения сообщения, указывающего, что приложение CRM загружено.

```
Console.WriteLine("CRM Application Starting");
```

```
Task loadUserDataTask = new Task(() =>
{
    Console.WriteLine("Loading User Data");
    Thread.Sleep(2000);
    Console.WriteLine("User data loaded");
});
Task loadCustomerDataTask = new Task(() =>
{
```

```

    Console.WriteLine("Loading Customer Data");
    Thread.Sleep(2000);
    Console.WriteLine("Customer data loaded");
});
loadUserDataTask.Start();
loadCustomerDataTask.Start();
Console.WriteLine("CRM Application Loaded");
Console.ReadLine();
loadUserDataTask.Dispose();
loadCustomerDataTask.Dispose();

```

Вышеприведенный код показывает общий шаблон как для параллельных, так и для многопоточных приложений. Код будет работать в потоке пользовательского интерфейса (UI), который может блокировать ввод от пользователя и создавать впечатление, что программа перестала отвечать. Путем запуска задач поиска данных в отдельных потоках мы позволяем программе быстрее реагировать, прежде чем информация будет получена. На выходе видно, что приложение загружается и готово к использованию, в то время как доступ к данным продолжается в фоновом режиме.

6.7. Ожидание завершения параллельных задач

Когда вы разрабатываете программное обеспечение, которое использует класс **Task** часто возникают ситуации, когда задача должна быть выполнена до того, как основной поток сможет продолжить обработку. Это может быть связано с тем, что параллельная задача генерирует результаты, которые необходимы позже в процессе, и вы должны дождаться, когда результаты будут доступны до того, как вы попытаетесь их использовать. Поэтому параллельные задачи должны быть синхронизированы для правильной работы программного обеспечения.

Параллельная библиотека задач (TPL) включает несколько методов, которые позволяют дождаться завершения одной или нескольких параллельных задач. Часто вам нужно дождаться завершения одной задачи, либо успешно, либо с исключением, прежде чем продолжить выполнение программы. Этого можно достичь с помощью метода **Task.Wait()**. В самой базовой ситуации этот метод блокирует поток, из которого он вызывается. Чтобы продемонстрировать использование метода **Wait**, сначала рассмотрим следующий пример кода. Здесь мы запускаем задачу, которая имитирует получение некоторых числовых данных из внешнего источника данных. Задача показывает сообщение, потом будет пауза в течение пяти секунд, а затем возвращает массив из десяти целых чисел. Основной поток использует результаты задачи, суммируя их и выводя результат.

```
int[] values = null;
Task loadDataTask = new Task(() =>
{
    Console.WriteLine("Loading data...");
    Thread.Sleep(5000);
    values = Enumerable.Range(1,10).ToArray();
});
loadDataTask.Start();
loadDataTask.Wait();
loadDataTask.Dispose();
Console.WriteLine("Data total = {0}", values.Sum());
```

Если есть вероятность слишком долгого выполнения задачи, вы можете предоставить значение таймаута, используя перегруженную версию метода **Wait**. Тайм-аут может быть предоставлен как целое число, которое представляет собой миллисекунды или как значение **TimeSpan**. Если задача завершается до истечения таймаута, основной поток разблокируется, как и

обычно, и метод **Wait** возвращает **true**. Если задача не завершается вовремя, основной поток разблокируется, и метод возвращает **false**. Однако это не останавливает параллельную задачу, которая может завершиться позднее.

6.8. Задачи продолжения

Когда вы пишете программу, которое выполняет задачи, которые выполняются параллельно, обычно существует несколько параллельных задач, которые зависят от результатов других. Эти задачи не должны запускаться до тех пор, пока не будут выполнены более ранние задачи. Задачи продолжения обычно создаются с использованием метода **ContinueWith** существующего экземпляра **Task**. Этот метод принимает единственный параметр, который определяет задачу, которая будет выполнена после завершения предыдущей задачи. Синтаксис метода **ContinueWith** следующий:

```
Task continuation = firstTask.ContinueWith(antecedent => { /* functionality */ });
```

Давайте создадим наш первый пример. В приведенном ниже коде мы моделируем два чтения данных, из базы данных или другого хранилища данных. Первая задача имитирует чтение пользовательских данных, чтобы получить идентификатор пользователя, который хранится в переменной **userID**. Вторая задача –это задача-продолжение. Она имитирует загрузку информации о разрешении пользователя, используя идентификатор, полученный в антецедентной задаче. Метод **Wait** необходим чтобы обе задачи выполнены, прежде чем будет показано последнее сообщение.

Когда вы запускаете код, вы можете видеть, что **loadUserPermissionsTask** не запускается до завершения загрузки **loadUserDataTask**.

```
string userID = null;  
var loadUserDataTask = new Task(() =>  
{  
    Console.WriteLine("Loading User Data");  
    Thread.Sleep(2000);
```

```

    userID = "1234";
    Console.WriteLine("User data loaded");
});
var loadUserPermissionsTask = loadUserDataTask.ContinueWith(t =>
{
    Console.WriteLine("Loading User Permissions for user {0}", userID);
    Thread.Sleep(2000);
    Console.WriteLine("User permissions loaded");
});
loadUserDataTask.Start();
loadUserPermissionsTask.Wait();
Console.WriteLine("CRM Application Loaded");
loadUserDataTask.Dispose();
loadUserPermissionsTask.Dispose();

```

6.9. Отмена выполнения задач

Когда выполняется длительный процесс в однопоточной программе, обычно его просто отменить. В параллельных приложениях, где одновременно могут работать несколько потоков, может быть гораздо сложнее скоординировать отмену нескольких связанных процессов. Например, в приложении Windows у вас может быть несколько задач, каждый из которых выполняет операцию с файлом. Если пользователь нажимает кнопку «Отмена», вы должны отменить все эти задачи, гарантируя, что данные не будут потеряны, а все файлы будут правильно закрыты и их объекты будут удалены.

TPL упрощает отмену задачи с помощью токенов отмены задач. Токен указывается при создании задачи. Если необходимо передать сигнал с другого процесса, требующий отмены, и этот запрос передается через токен. Один и тот же токен отмены может использоваться несколькими параллельными задачами, позволяя одному запросу аннулирования прекратить выполнение любого количества задач. Сам токен является

экземпляром структуры **CancellationToken**. Токен не создается с помощью конструктора, сначала необходимо создать экземпляр класса **CancellationTokenSource** и получить токен из его свойства **Token**. При создании задач, которые поддерживают отмену, необходимо передать токен в конструктор задачи в дополнение к выполняемому делегату. Токен также должен быть доступен в пределах делегата, чтобы вы могли получить доступ к его свойствам и методам. Основное свойство это **IsCancellationRequested**, которое возвращает логическое значение, указывающее, была ли отменена.

Используя токен отмены вы должны периодически проверять, была ли задача отменена и в случае отмены выполнить необходимые действия перед выходом. Это может включать закрытие файлов или соединений с базой данных, завершение или откат транзакций и освобождение ресурсов.

Рассмотрим пример. В методе **Main** мы создаем экземпляр класса **CancellationTokenSource** и используем его для получения токена. Затем мы передаем этот токен нашему конструктору задачи **Task**. После запуска задачи, мы ждем пока пользователь не нажмет **Enter** для отмены задачи с помощью вызова метода **Cancel** объекта **CancellationTokenSource**. Метод **DoLongRunningTask** вызывается из параллельной задачи. Он имитирует длительный процесс. Перед запуском цикла проверяется свойство **IsCancellationRequested** маркера. Поскольку возможно, что задача могла быть отменена до того, как она действительно начала выполняться, эта проверка позволяет остановить ее, не выполняя никакой работы. Флаг **IsCancellationRequested** снова проверяется во время каждой итерации. Если **true**, отображается сообщение, и цикл завершается. Попробуйте запустить код и разрешить выполнение нескольких итераций до нажатия **Enter**, чтобы отменить задачу.

```
static void Main()
```

```
{
```

```

var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;
var task = new Task(() => DoLongRunningTask(token), token);
Console.WriteLine("Press Enter to cancel");
task.Start();
Console.ReadLine();
tokenSource.Cancel();
task.Wait();
task.Dispose();
Console.WriteLine("Press Enter to exit");
Console.ReadLine();
}

static void DoLongRunningTask(CancellationToken token)
{
    if (token.IsCancellationRequested)
    {
        Console.WriteLine("Cancelled before long running task started");
        return;
    }
    for (int i = 0; i <= 100; i++)
    {
        Console.WriteLine("{0}%", i);
        Thread.Sleep(1000);
        if (token.IsCancellationRequested)
        {
            Console.WriteLine("Cancelled");
            break;
        }
    }
}

```



```
}  
}
```

Во многих ситуациях вы запускаете несколько параллельных задач. Когда требуется отмена выполнения задач, вы можете отменить группу задач, а не только одну. Это достигается за счет использования одного и того же токена для каждой задачи в группе. Если же у вас есть несколько групп задач, которые можно отменить, используйте отдельный файл **CancellationTokenSource** для генерации токенов для каждой группы.

6.10. Параллельный LINQ

Language-Integrated Query (**LINQ**) предоставляет декларативную модель, которая позволяет запрашивать последовательности данных, таких как коллекции в памяти, документы XML и данные базы данных. Характер многих запросов означает, что их можно легко распараллелить. Большинство запросов выполняют одну и ту же группу действий для каждого элемента в коллекции. Если все эти действия являются независимыми, без побочных эффектов, вызванных порядком, в котором они появляются, вы часто можете добиться большого увеличения производительности за счет разделения работы между несколькими процессорными ядрами.

Чтобы поддержать эти сценарии .NET Framework версии 4.0 представила **Parallel LINQ (PLINQ)**. **PLINQ** предоставляет те же стандартные операторы запросов и синтаксис выражений запроса, что и **LINQ**. Основное различие заключается в том, что исходные данные могут быть разбиты на части с использованием декомпозиции данных. Эти меньшие группы данных затем потенциально обрабатываются всеми доступными ядрами ЦП. **PLINQ** имеет некоторые ограничения, которые означают, что он не является прямой заменой **LINQ** и не может быть параметром по умолчанию для запросов. Ключом к числу ограничений является то, что побочные эффекты обработки отдельных элементов из исходных последовательностей, такие как результат обработки одного элемента, зависящего от другого, могут приводить к непредсказуемым

результатам. Это связано с тем, что исходные элементы обычно не обрабатываются в исходном порядке. Второе ограничение заключается в том, что PLINQ обеспечивает параллелизм для данных в памяти, таких как коллекции или предварительно загруженный XML.

Чтобы показать, как может быть изменен запрос LINQ для параллельной обработки, сначала нужна последовательная версия. В приведенном ниже коде показан очень простой запрос. Здесь мы начинаем с массива, содержащего целые числа от одного до десяти. Используя оператор **Select**, мы проецируем его на новую последовательность, содержащую квадраты исходных значений. Поскольку LINQ использует отложенное выполнение, новая последовательность не генерируется до тех пор, пока данные не будут доступны. Это означает, что цикл `foreach` заставляет запрос выполняться и выводит результаты.

```
int[] sequence = Enumerable.Range(1, 10).ToArray();
var squares = sequence.Select(x => x * x);
foreach (var square in squares)
{
    Console.WriteLine(square + " ");
}
Console.ReadLine();
```

Результат: 1 4 9 16 25 36 49 64 81 100

LINQ работает с последовательностями, реализующими интерфейс **IEnumerable<T>**. Чтобы обозначить, что мы хотим использовать PLINQ, мы должны использовать статический метод **AsParallel**. Это метод расширения **IEnumerable<T>**, поэтому его можно применять к любой последовательности, поддерживающей операции LINQ. Он возвращает объект типа **ParallelQuery <T>**. Когда у вас есть параллельная последовательность данных, вы можете использовать ее в качестве источника для операций LINQ,

как и в любой другой последовательности. **PLINQ** разлагает данные таким образом, чтобы обеспечить эффективную параллельную обработку.

Чтобы распараллелить запрос, добавьте вызов в **AsParallel**, как показано ниже:

```
var squares = sequence.AsParallel().Select(x => x * x);
```

Результат: 100 1 25 36 49 64 81 4 9 16

Возможно, вы заметили, что результаты **PLINQ** запроса правильны, но они появились в другом порядке, чем при использовании последовательной версии **LINQ**. Это побочный продукт при параллельном выполнении запросов **LINQ**. В некоторых случаях упорядочение результатов не имеет значения, особенно если данные впоследствии сортируются с использованием стандартного оператора запроса **OrderBy** или каким-либо императивным методом. В других случаях нарушения первоначального порядка может быть катастрофической. В таких ситуациях вам нужно сохранить упорядочение результатов в соответствии с порядком ввода. **PLINQ** поддерживает сохранение порядка параллельного источника данных с использованием метода **AsOrdered**. Этот метод следует использовать только тогда, когда необходимо поддерживать порядок результатов, так как это может значительно снизить производительность ваших запросов.

Следующий запрос использует метод **AsOrdered** после **AsParallel** для сохранения порядка результатов.

```
var squares = sequence.AsParallel().AsOrdered().Select(x => x * x);
```

Результат: 1 4 9 16 25 36 49 64 81 100

Когда вы выполняете последовательный запрос с использованием **LINQ**, любой обработанный элемент данных может привести к исключению. Когда исключение брошено, запрос немедленно останавливается выполнение. С **PLINQ** возможно одновременное выполнение нескольких операций. Если одно из них выдает исключение, все остальные потоки останавливаются, но только после завершения запланированных операций. Это может означать задержку

между исключительным событием и остановкой запроса **PLINQ**, если операции запроса медленны. Это также означает, что любая из других параллельных операций может также генерировать исключение.

Чтобы устранить возможность запроса, вызывающего множественные исключения, все исключения из запроса **PLINQ** объединяются в одно исключение **AggregateException**, которое бросается, когда все потоки выполнения останавливаются. Как и при использовании параллельных циклов и задач, вы можете перехватить это исключение и изучить его свойство **InnerExceptions**, чтобы найти все исключения.

7. MPI

7.1. Введение в MPI

К 1994 году был определен полный интерфейс и стандарт (MPI-1). MPI - это только определение интерфейса. Затем разработчикам приходилось создавать реализации интерфейса для своих соответствующих архитектур. После того, как были созданы первые его реализации, MPI был широко принят и по-прежнему остается методом написания приложений, передающих сообщения. Рассмотрим классические концепции MPI модели передачи сообщений в параллельном программировании.

Первая концепция - понятие **коммуникатора**. Коммуникатор определяет группу процессов, которые имеют возможность связываться друг с другом. В этой группе процессов каждому процессу присваивается уникальный **ранг**. Основа коммуникации построена на процессах отправки и получения сообщений между процессами. Процесс может отправить сообщение другому процессу, предоставив **ранг** процесса и уникальный **тег** для идентификации сообщения. Затем получатель может получить сообщение с заданным тегом, а затем обрабатывать данные соответствующим образом. Такой тип коммуникации, который связывает одного отправителя с одним получателем, называется «**точка-точка**». Существует много случаев, когда процессам может

потребуется общение со всеми остальными. Например, когда мастер-процесс должен передавать информацию всем своим рабочим процессам. В этом случае было бы громоздко писать код, который выполняет все отправки и принимает. MPI может обрабатывать широкий спектр этих типов **коллективных** сообщений, которые включают в себя все процессы.

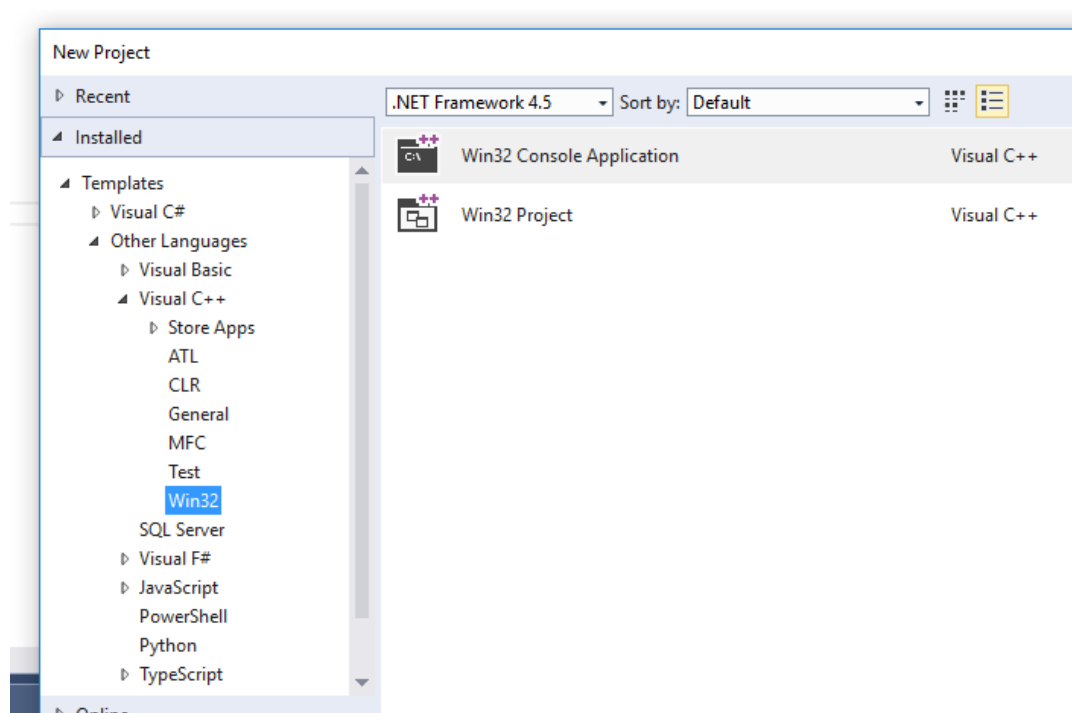
7.2. Начало работы с MPI с помощью Visual Studio 2013

Шаг 1: Загрузите и установите SDK HPC Pack 2008

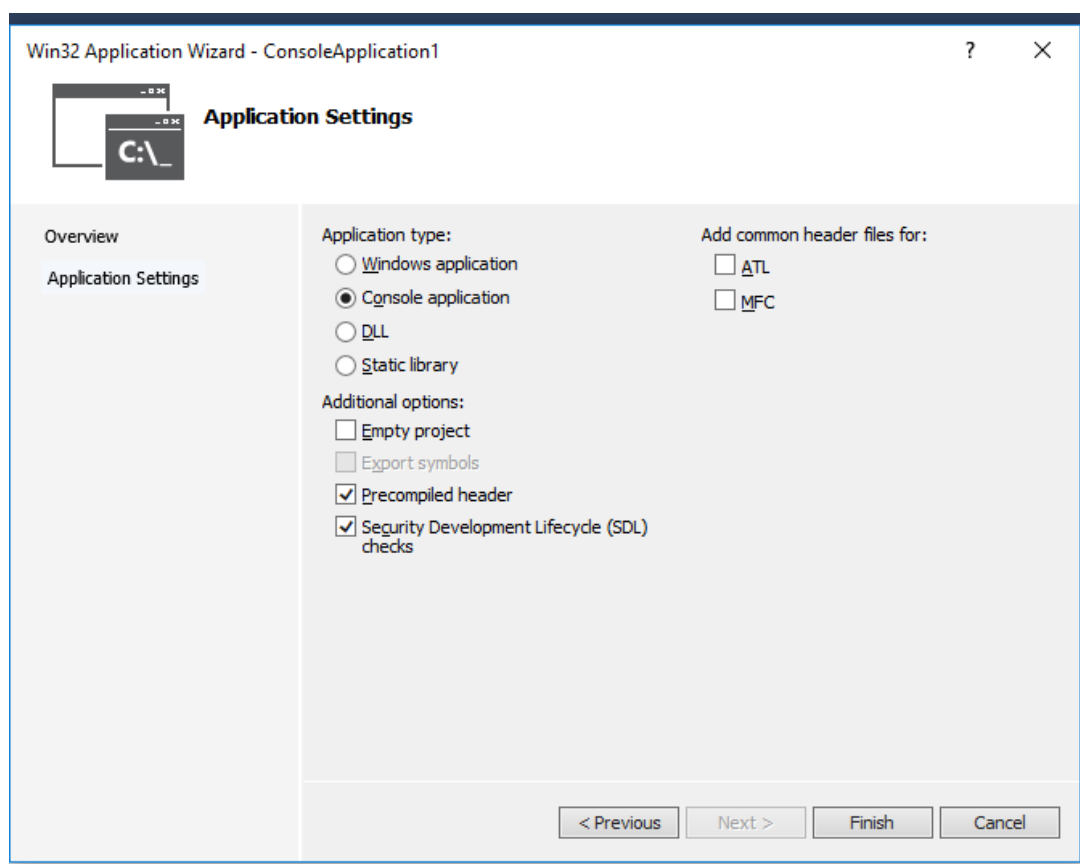
Необходимо установить пакет HPC Pack 2008 SDK SP2 (в вашем случае может быть уже другая версия), доступный на официальном сайте Microsoft. Разрядность пакета и системы должны соответствовать.

Шаг 2: Создание проекта

Откройте Visual Studio, откройте меню «Файл» и выберите «Создать»> «Проект». Появится диалоговое окно:



После названия вашего проекта нажмите «ОК».

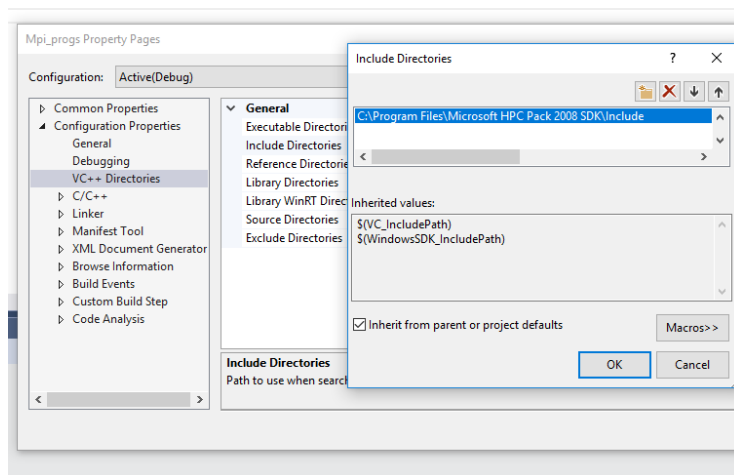


Выключите опцию **Precompiled Header**. Это, как правило, упрощает компиляцию исходного кода платформ, отличных от Windows.

Шаг 3. Настройте проект для запуска MPI.

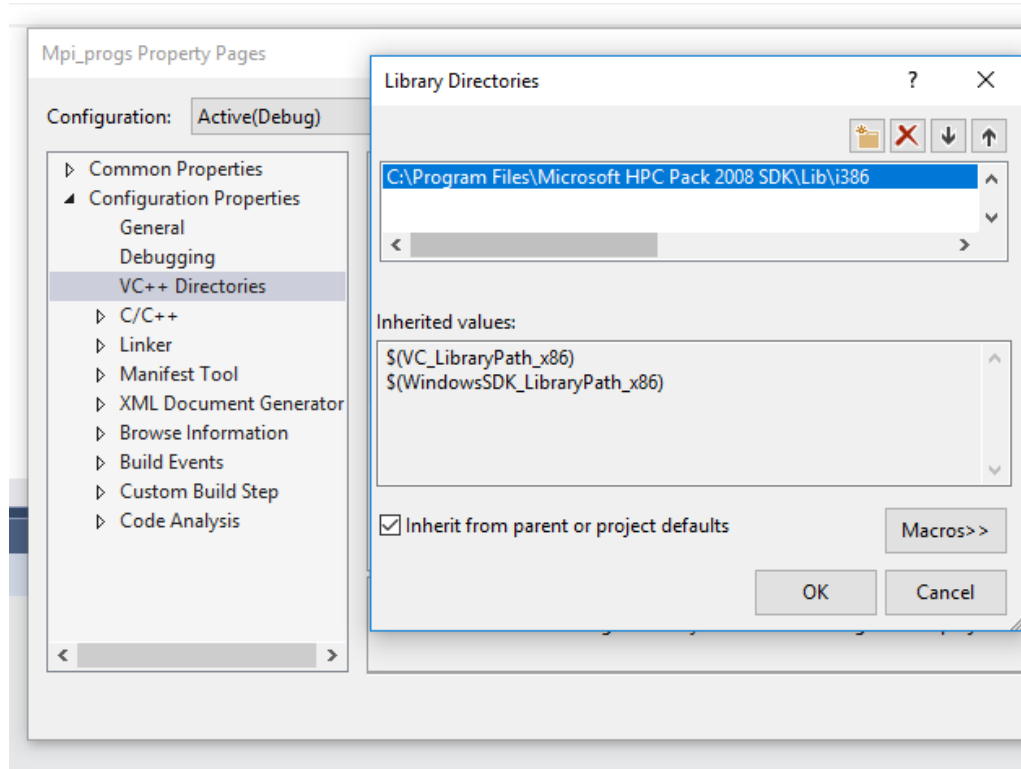
Теперь, когда вы работаете с программой, мы должны настроить вашу программу на включение библиотек MPI и файлов заголовков. Щелкните правой кнопкой мыши свой проект в окне обозревателя решений и выберите «Свойства». Далее во вкладке VC++ Directories необходимо прописать в поле Include Directories:

“C:\Program Files\Microsoft HPC Pack 2008 SDK\Include”



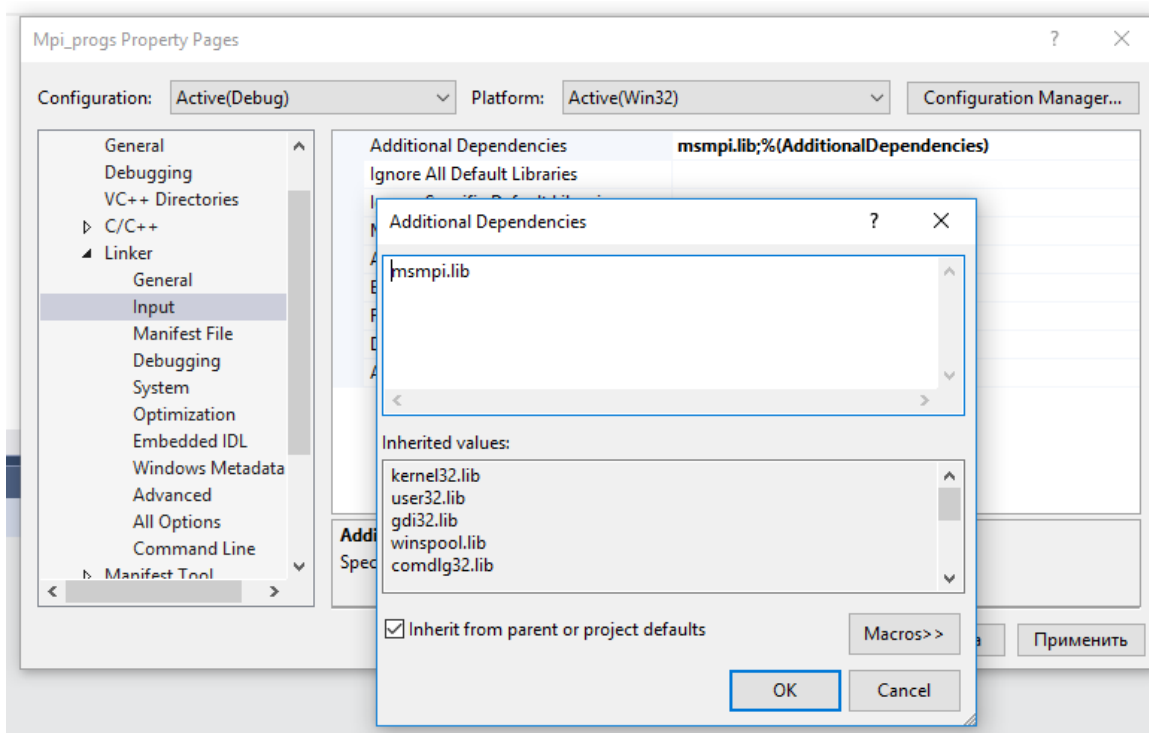
В поле Library Directories:

“C:\Program Files\Microsoft HPC Pack 2008 SDK\Lib\amd64”



Далее во вкладке **Linker – Input** в поле **Additional Dependencies** необходимо указать библиотеку

msmpi.lib



Шаг 4: Введите следующий программный код с использованием MPI

```
#include <mpi.h>
```

```
#include <stdio.h>
```

```
int main(int argc, char** argv) {
```

```
    MPI_Init(NULL, NULL);
```

```
    int world_size;
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

```
    int world_rank;
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

```
    char processor_name[MPI_MAX_PROCESSOR_NAME];
```

```
    int name_len;
```

```
    MPI_Get_processor_name(processor_name, &name_len);
```

```
    printf("Hello world from processor %s, rank %d out of %d processors\n",  
        processor_name, world_rank, world_size);
```

```
    MPI_Finalize();
```

```
}
```

Шаг 5: запуск программы с несколькими процессами

Если вы сейчас компилируете и запускаете программу, вы обнаружите, что программа работает только с 1 процессом, даже если у вас есть двухъядерный или четырехъядерный компьютер. Чтобы запустить MPI, вам нужно запустить программу через **mpiexec.exe**. Для этого перейдите в меню «Пуск» и выберите «Выполнить». Введите **cmd** и нажмите **enter**. Перейдите в каталог проекта используя команду **cd <имя каталога>**. Когда вы перейдете в каталог проекта, введите **mpiexec -n количество_потоков prog1.exe**. Указав значение ключа -n, вы можете заставить свою программу начинать с любого количества потоков. Например, если укажем 4 потока с ключом **-n 4**, то получим следующий вывод на консоль:

```
Hello world from processor DESKTOP-94N0G7Q, rank 3 out of 4 processors  
Hello world from processor DESKTOP-94N0G7Q, rank 2 out of 4 processors  
Hello world from processor DESKTOP-94N0G7Q, rank 0 out of 4 processors  
Hello world from processor DESKTOP-94N0G7Q, rank 1 out of 4 processors
```

7.3. Основные функции MPI

Рассмотрим более подробно код примера приведенный в предыдущем разделе. Первый шаг к созданию MPI-программы включает в себя файлы заголовков **MPI #include<mpi.h>**. После этого среда MPI должна быть инициализирована с помощью вызова функции:

MPI_Init(int* argc, char* argv)**

Вызов этой функции приводит к созданию всех глобальных и внутренних переменных MPI. Например, коммуникатор формируется вокруг всех процессов, которые были порождены, и каждому процессу присваиваются уникальные ранги. После вызова **MPI_Init** вызываются две функции которые используются почти в каждой программе MPI.

MPI_Comm_size(MPI_Comm communicator, int* size) -возвращает размер текущего коммуникатора. В нашем примере коммуникатор **MPI_COMM_WORLD** охватывает все процессы, поэтому этот вызов должен

возвращать количество процессов, которое было указано при запуске программы из командной строки.

MPI_Comm_rank(MPI_Comm communicator, int* rank) - возвращает ранг процесса в коммуникаторе. Каждому процессу внутри коммуникатора присваивается инкрементный ранг, начиная с нуля. Ранжирование процессов в основном используется для их идентификации при отправке и получении сообщений. Во многих программах используется также другая функция:

MPI_Get_processor_name(char* name, int* name_length), которая возвращает фактическое имя процессора, на котором выполняется процесс. Последний вызов в этой программе это:

MPI_Finalize()

MPI_Finalize используется для очистки среды MPI.

7.4. MPI Send and Receive

Отправка и получение сообщений - это две основные концепции MPI. Почти каждая функция MPI может быть реализована с помощью основных вызовов отправки и получения. Вызовы и прием MPI сообщений работают следующим образом. Процесс **A** упаковывает все необходимые данные в буфер для процесса **B**. После того, как данные упакованы в буфер, устройство связи (которое часто является сетью) отвечает за маршрутизацию сообщения в нужное место. Местоположение сообщения определяется рангом процесса. Иногда бывают случаи, когда процесс **A** может послать много разных типов сообщений **B**. Вместо того, чтобы **B** должен был пройти дополнительные меры для дифференциации этих сообщений, MPI позволяет отправителям и получателям также указывать идентификаторы сообщений с сообщением (известным как **теги**). Когда процесс **B** запрашивает только сообщение с определенным номером тега, сообщения с разными тегами будут буферизироваться сетью, пока **B** не будет готов для них. Рассмотрим MPI функций отправки и получения.

MPI_Send(void* data,int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator) – отправка сообщения

MPI_Recv(void* data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status* status) –получение сообщения

Первый аргумент (**data**) - это буфер данных, второй (**count**) и третий (**datatype**) аргументы описывают количество и тип элементов, которые находятся в буфере. **MPI_Send** отправляет точное количество элементов равное **count** и **MPI_Recv** будет получать **такое же** количество элементов. Четвертый и пятый аргументы определяют ранг процесса отправки (**source**) и получения и тега (**tag**) сообщения. Шестой аргумент указывает коммуникатор, а последний аргумент (только для **MPI_Recv**) предоставляет информацию о полученном сообщении.

7.5. Элементарные типы данных MPI

Функции **MPI_Send** и **MPI_Recv** используют типы данных MPI в качестве средства для определения структуры сообщения на более высоком уровне. Например, если процесс хочет отправить одно целое число другому, он будет использовать тип данных **MPI_INT**. Другие элементарные типы данных MPI перечислены ниже с их эквивалентными типами данных C.

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int

MPI datatype	C equivalent
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

В качестве примера рассмотрим программу для пинг-понга. В этом примере процессы используют функции **MPI_Send** и **MPI_Recv** и постоянно посылают сообщения друг другу, пока они не решат остановиться. Основные части кода программы выглядят так.

```
int main(int argc, char** argv) {
    const int PING_PONG_LIMIT = 10;
    MPI_Init(NULL, NULL);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    if (world_size != 2) {
        fprintf(stderr, "World size must be two for %s\n", argv[0]);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    int ping_pong_count = 0;
    int partner_rank = (world_rank + 1) % 2;
    while (ping_pong_count < PING_PONG_LIMIT) {
        if (world_rank == ping_pong_count % 2) {
            ping_pong_count++;
            MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0,
MPI_COMM_WORLD);
            printf("%d sent and incremented ping_pong_count %d to %d\n",
```

```

        world_rank, ping_pong_count, partner_rank);
    }
    else {
        MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank, 0,
MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
        printf("%d received ping_pong_count %d from %d\n",
        world_rank, ping_pong_count, partner_rank);
    }
}
MPI_Finalize();
}

```

Этот пример предназначен для выполнения только на двух процессах, если вы указали большее или меньшее количество процессов, то программа будет завершена вызовом метода **MPI_Abort**. Переменная **ping_pong_count** инициализируется нулем и увеличивается на каждом этапе пинг-понга посредством процесса отправки. По мере того, как значение **ping_pong_count** увеличивается, процессы по очереди становятся отправителем и получателем. Наконец, после достижения лимита процессы прекращают отправку и получение.

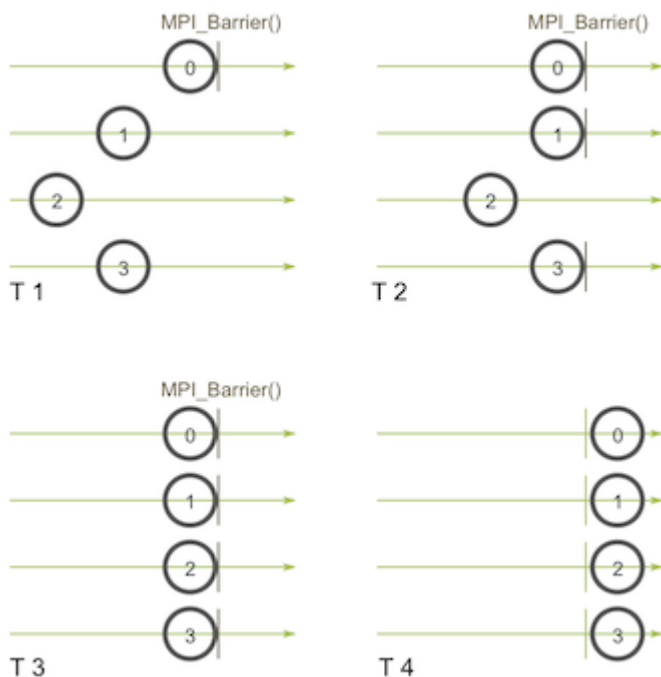
7.6. Коллективные коммуникации в MPI

Выше мы рассматривали коммуникации типа «точка-точка», которая представляет собой связь между двумя процессами. В этом разделе рассмотрим **коллективные коммуникации**. Коллективная коммуникация - это способ коммуникации, который включает участие всех процессов в коммуникаторе. Одна из вещей, которые следует помнить о коллективной коммуникации, заключается в том, что она подразумевает **точку синхронизации** между процессами. Это означает, что все процессы должны достигнуть точки в своем коде, прежде чем они смогут снова начать

выполнение. MPI имеет специальную функцию, предназначенную для синхронизации процессов:

MPI_Barrier(MPI_Comm communicator)

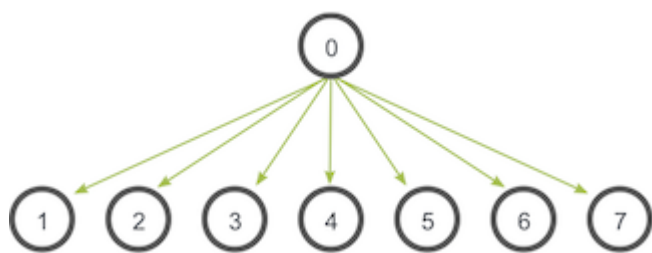
Название функции довольно описательно - функция образует барьер, и никакие процессы в коммуникаторе не могут пройти через барьер, пока все они не вызовут функцию. Здесь приведена иллюстрация использования этой функции. На рисунке горизонтальная ось представляет собой выполнение программы, а круги представляют собой различные процессы:



MPI_Barrier может быть полезным для многих вещей. Одним из основных применений **MPI_Barrier** является синхронизация программы, так что части параллельного кода могут быть синхронизированы по времени.

Широковещательная передача является одним из стандартных коллективных методов передачи сообщений в MPI. Во время трансляции один процесс отправляет одни и те же данные ко всем процессам в коммуникаторе. Одним из основных видов использования широковещательной передачи является отправка пользовательского ввода в параллельную программу или отправка параметров конфигурации для всех процессов.

Схема широковещательной передачи выглядит так:



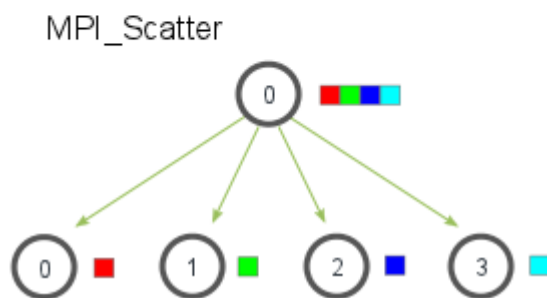
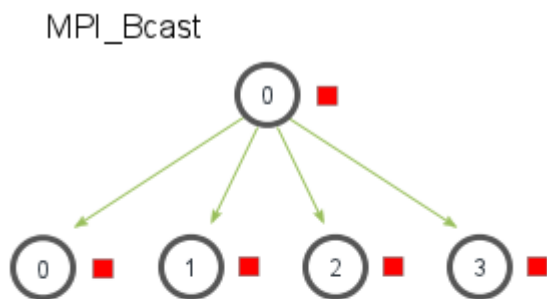
В этом примере нулевой процесс является **корневым** процессом и имеет исходную копию данных. Все остальные процессы получают копию данных. В MPI вещание может быть выполнено с использованием **MPI_Bcast**. Прототип функции выглядит следующим образом:

```
MPI_Bcast( void* data, int count, MPI_Datatype datatype, int root, MPI_Comm communicator)
```

Корневой процесс и процессы получатели называют одну и ту же функцию **MPI_Bcast**. Если в корневом процессе была вызвана функция **MPI_Bcast**, переменная **data** будет отправлена всем другим процессам. Когда все процессы приемники вызывают **MPI_Bcast**, переменная **data** будет заполнена данными из корневого процесса.

7.7. Функции Scatter, Gather и Allgather

MPI_Scatter -это функция коллективной передачи данных, которая очень похожа на **MPI_Bcast**. При вызове **MPI_Scatter** назначенный корневой процесс также отправляет данные ко всем процессам в коммуникаторе. Основное различие между **MPI_Bcast** и **MPI_Scatter** заключается в том, что **MPI_Bcast** отправляет один и тот же фрагмент данных всем процессам, в то время как **MPI_Scatter** отправляет разные куски массива на разные процессы.



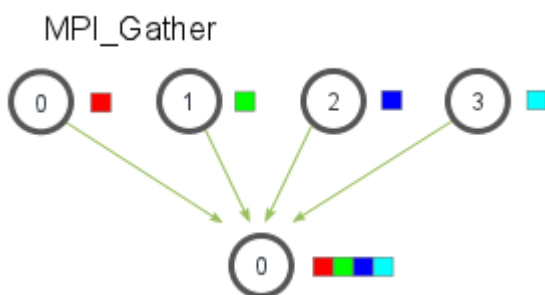
На иллюстрации показано что **MPI_Bcast** берет один элемент данных в корневом процессе (красное поле) и копирует его ко всем другим процессам. **MPI_Scatter** принимает массив элементов и распределяет элементы в порядке ранга процесса. Первый элемент (в красный) переходит в нулевой процесс, второй элемент (зеленый) переходит в первый процесс и тд. Хотя корневой процесс (нулевой процесс) содержит весь массив данных, **MPI_Scatter** скопирует соответствующий элемент в буфер приема процесса. Вот как выглядит прототип функции **MPI_Scatter**.

```
MPI_Scatter( void* send_data, int send_count, MPI_Datatype
send_datatype, void* recv_data, int recv_count, MPI_Datatype
recv_datatype, int root, MPI_Comm communicator)
```

Первый параметр **send_data**- массив данных, который находится на корневом процессе. Второй и третий параметры, **send_count** и **send_datatype**, указывают, сколько элементов определенного типа данных MPI будет отправлено каждому процессу. На практике **send_count** часто равен количеству элементов в массиве, деленному на количество процессов. Параметр **recv_data** представляет собой буфер данных, который может содержать **recv_count** элементы, которые имеют тип данных **recv_datatype**. Последние параметры **root** и **communicator** указывают

корневой процесс, который раздает массив данных и коммуникатор, в котором находятся процессы.

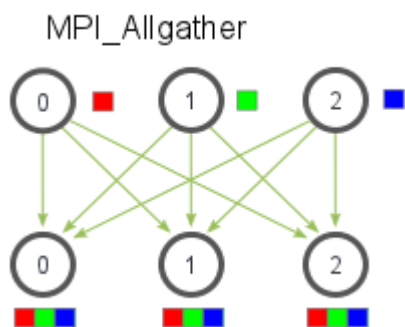
Функция **MPI_Gather** является обратной функции **MPI_Scatter**. Вместо того, чтобы раздавать элементы из одного процесса во многие процессы, он **MPI_Gather** принимает элементы из многих процессов и собирает их в один процесс. Эта процедура очень полезна для многих параллельных алгоритмов, таких как параллельная сортировка и поиск. Ниже приведена простая иллюстрация этого алгоритма.



Подобно **MPI_Scatter**, **MPI_Gather** принимает элементы от каждого процесса и собирает их в корневой процесс. Элементы упорядочены по рангу процесса, из которого они были получены. Прототип функции **MPI_Gather** идентичен **MPI_Scatter**.

MPI_Gather(void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype recv_datatype, int root, MPI_Comm communicator)

MPI_Scatter, **MPI_Gather** являются функциями, которые выполняют шаблоны передачи данных «один к одному» или «один ко многим», что просто означает, что многие процессы отправляют, а получает один процесс или наоборот. Зачастую полезно иметь возможность отправлять многие элементы во многие процессы (т. е. шаблон передачи «многие ко многим»). Для этого в MPI имеется функция **MPI_Allgather**. На следующем рисунке показано, как данные распределяются после вызова **MPI_Allgather**.



Так же, как и при вызове функции **MPI_Gather** элементы из каждого процесса собираются в порядке их ранга, за исключением того, что элементы будут собраны во все процессы. Объявление функции **MPI_Allgather** почти идентично **MPI_Gather**, разница лишь в отсутствии корневого процесса.

```
MPI_Allgather( void* send_data, int send_count, MPI_Datatype  
send_datatype, void* recv_data, int recv_count, MPI_Datatype  
recv_datatype, MPI_Comm communicator)
```

В качестве примера рассмотрим программу, которая вычисляет среднее значение массива. Хотя программа довольно проста, она демонстрирует, как можно использовать MPI для разделения работы между процессами, выполнения вычислений на подмножествах данных, а затем агрегировать меньшие фрагменты в окончательный ответ.

```
float *create_rand_nums(int num_elements) {  
  
    float *rand_nums = (float *)malloc(sizeof(float) * num_elements);  
  
    assert(rand_nums != NULL);  
  
    int i;  
  
    for (i = 0; i < num_elements; i++) {  
  
        rand_nums[i] = (rand() / (float)RAND_MAX);  
  
    }  
  
    return rand_nums;  
  
}
```

```

float compute_avg(float *array, int num_elements) {

    float sum = 0.f;    int i;

    for (i = 0; i < num_elements; i++) {

        sum += array[i];

    }

    return sum / num_elements;

}

int main(int argc, char** argv) {

    int num_elements_per_proc = 10000000;

    srand(time(NULL));

    MPI_Init(NULL, NULL);

    int world_rank;

    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    int world_size;

    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    float *rand_nums = NULL;

    if (world_rank == 0) {

        rand_nums = create_rand_nums(num_elements_per_proc *
world_size);

    }

    float *sub_rand_nums = (float *)malloc(sizeof(float) *
num_elements_per_proc);

    MPI_Scatter(rand_nums, num_elements_per_proc, MPI_FLOAT,
sub_rand_nums, num_elements_per_proc, MPI_FLOAT, 0,
MPI_COMM_WORLD);

```

```

    float sub_avg = compute_avg(sub_rand_nums,
num_elements_per_proc);

    float *sub_avgs = (float *)malloc(sizeof(float) * world_size);

    MPI_Allgather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT,
MPI_COMM_WORLD);

    float avg = compute_avg(sub_avgs, world_size);

    printf("Avg of all elements from proc %d is %f\n", world_rank, avg);

    if (world_rank == 0) {

        free(rand_nums);

    }

    free(sub_avgs);

    free(sub_rand_nums);

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Finalize();
}

```

Здесь функция **create_rand_nums** заполняет массив случайными числами, а функция **compute_avg** вычисляет среднее значение. Если вы запустите программу, то результат должен выглядеть примерно так:

Avg of all elements from proc 1 is 0.499927

Avg of all elements from proc 0 is 0.499927

Обратите внимание, что цифры генерируются случайным образом, поэтому ваш конечный результат может отличаться от приведенного ниже.

7.8. Функции MPI Reduce and Allreduce

Редукция - это классическая концепция функционального программирования. Редукция данных включает в себя сокращение набора чисел в меньший набор чисел с помощью какой-либо функции. Например, допустим, у нас есть список чисел [1, 2, 3, 4, 5]. Редукция этого списка чисел с помощью функции **sum** будет **15**. а сокращение умножения **multiply** будет **120**.

В MPI есть удобная функция, **MPI_Reduce** которая выполняет почти все общие сокращения, с которыми программист может столкнуться при разработке параллельного приложения. Подобно **MPI_Gather**, **MPI_Reduce** эта функция принимает массив входных элементов для каждого процесса и возвращает массив выходных элементов в корневой процесс. Выходные элементы содержат уменьшенный результат. Прототип **MPI_Reduce** выглядит так:

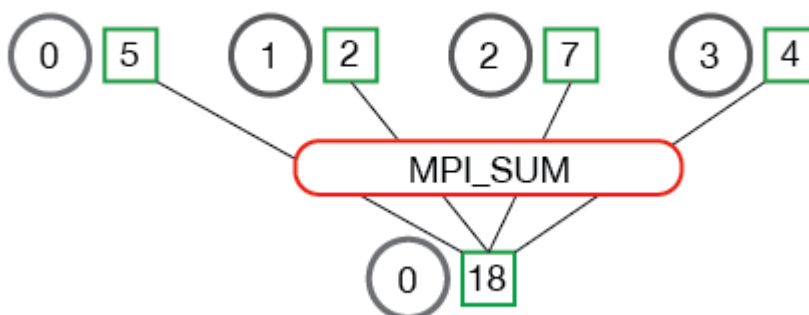
MPI_Reduce(void* send_data, void* recv_data, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm communicator)

В отличие от уже известных нам функции функция **MPI_Reduce** содержит параметр **MPI_Op op**, указывающий операцию, которая будет применена к данным. MPI содержит обширный набор общих операций редукции, вот некоторые наиболее часто используемые из них:

- **MPI_MAX** - Возвращает максимальный элемент.
- **MPI_MIN** - Возвращает минимальный элемент.
- **MPI_SUM** - Суммирует элементы.
- **MPI_PROD** - Умножает все элементы.
- **MPI_LAND**- Выполняет логический оператор И ко всем элементам.
- **MPI_LOR**- Выполняет логический оператор ИЛИ ко всем элементам.

Ниже приведена иллюстрация выполнения **MPI_Reduce**.

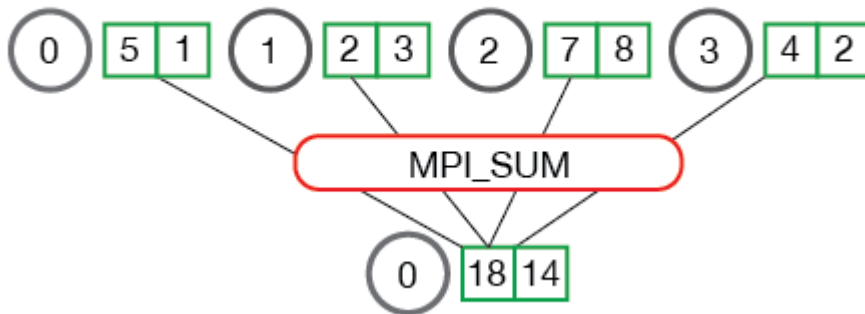
MPI_Reduce



В этом примере каждый процесс содержит одно целое число. **MPI_Reduce** вызывается с корневым процессом 0 и использует **MPI_SUM** в качестве операции редукции. Четыре числа суммируются и

результатом сохраняется в корневом процессе. Также полезно понимать, что происходит, когда процессы содержат несколько элементов. На приведенном ниже рисунке показано редукция данных на один процесс.

MPI_Reduce



Результирующее суммирование происходит на основе каждого элемента. Другими словами, вместо суммирования всех элементов из всех массивов в один элемент i -й элемент из каждого массива суммируется в i -м элементе массива результатов процесса 0.

Рассмотрим пример использования функции **MPI_Reduce**. В предыдущем разделе мы изучили как вычислять среднее значение с помощью функций **MPI_Scatter** и **MPI_Gather**. Использование **MPI_Reduce** упрощает намного код, здесь приведена только основная часть кода.

```

float *rand_nums = NULL;

rand_nums = create_rand_nums(num_elements_per_proc);

float local_sum = 0; int i;

for (i = 0; i < num_elements_per_proc; i++) {
    local_sum += rand_nums[i];
}

printf("Local sum for process %d - %f, avg = %f\n", world_rank,
local_sum, local_sum / num_elements_per_proc);

float global_sum;
  
```

```
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0,
MPI_COMM_WORLD);
```

```
if (world_rank == 0) {
```

```
    printf("Total sum = %f, avg = %f\n", global_sum,
```

```
        global_sum / (world_size * num_elements_per_proc));
```

```
}
```

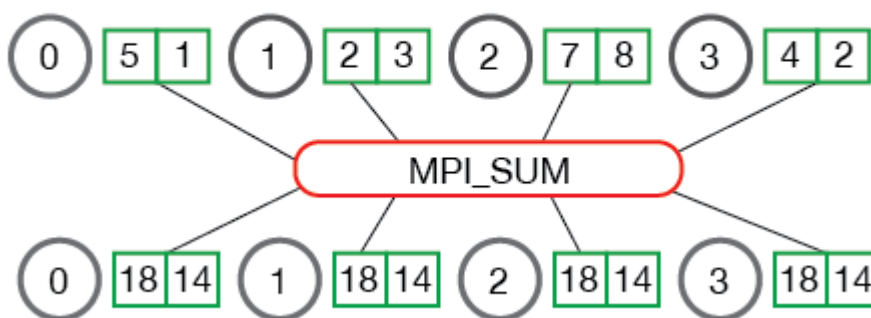
В приведенном выше коде каждый процесс создает случайные числа и вычисляет локальную сумму **local_sum**. Затем **local_sum** он сводится к корневому процессу с использованием **MPI_SUM**.

Многим параллельным приложениям потребуется доступ к уменьшенным результатам во всех процессах, а не только в корневом процессе. Для этого используется функция **MPI_Allreduce**:

```
MPI_Allreduce( void* send_data, void* recv_data, int count, MPI_Datatype  
datatype, MPI_Op op, MPI_Comm communicator)
```

Ниже проиллюстрирована структура выполнения **MPI_Allreduce**.

MPI_Allreduce



MPI_Allreduce идентичен **MPI_Reduce**, отличие только в том, что ему не нужен идентификатор корневого процесса, так как результаты распределяются по всем процессам.

7.9. Группы и коммуникаторы в MPI

Как мы видели MPI позволяет одновременно передавать информацию между всеми процессами в коммуникаторе, чтобы делать такие вещи, как

распространение данных из одного процесса во другие процессы с использованием **MPI_Scatter** или выполнение редукции данных с использованием **MPI_Reduce**. Во всех предыдущих разделах мы использовали коммуникатор **MPI_COMM_WORLD**. Для простых приложений этого достаточно, так как мы имеем относительно небольшое количество процессов. Однако, когда приложения начинают увеличиваться, это становится менее практичным, и для более сложных случаев может оказаться полезным иметь больше коммуникаторов. В этом разделе мы покажем, как создавать новые коммуникаторы для связи с подмножеством исходной группы процессов одновременно. Самая общая функция, используемой для создания новых коммуникаторов:

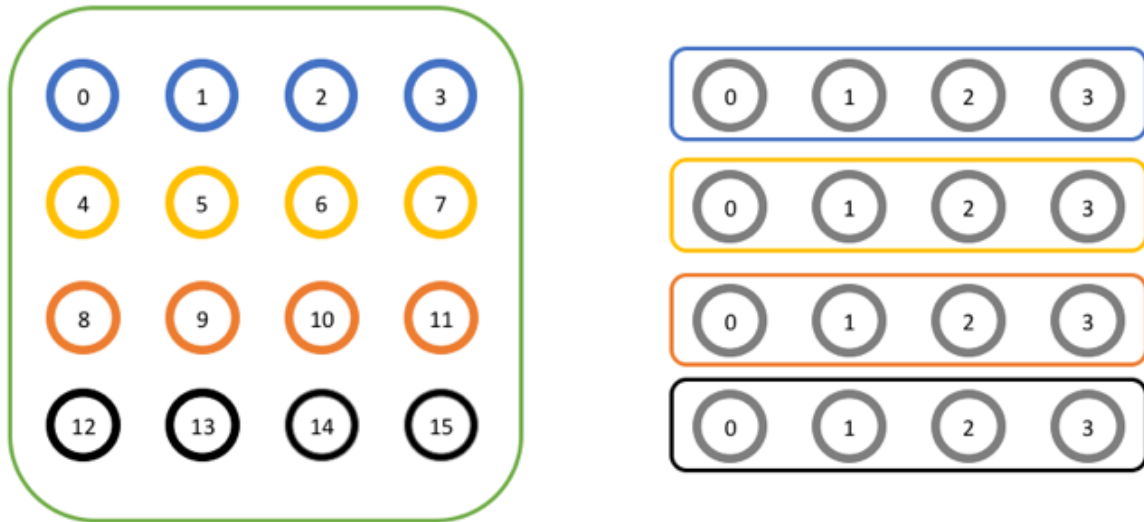
MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm* newcomm)

Эта функция создает новые коммуникаторы путем «расщепления» коммуникатора в группу субкоммуникаторов на основе входных значений **color** и **key**. Первый аргумент **comm** – это коммуникатор, который будет использоваться в качестве основы для новых коммуникаторов. Это может быть **MPI_COMM_WORLD**, но это может быть и любой другой коммуникатор. Второй аргумент, **color** определяет, к какому новому коммуникатору будут принадлежать все процессы. Все процессы, которые передают одно и то же значение **color**, назначаются одному и тому же коммуникатору. Третий аргумент **key**, определяет ранг в каждом новом коммуникаторе. Процесс, который с наименьшим значением **key** будет иметь ранг 0, следующим наименьшим будет ранг 1 и т. Д. Последним аргументом **newcomm** является ссылка на возвращаемый новый коммуникатор.

Рассмотрим простой пример, когда мы пытаемся разделить один глобальный коммуникатор на набор более мелких коммуникаторов. В этом примере мы предположим, что мы логически выложили наш оригинальный коммуникатор в сетку 4x4 из 16 процессов, и мы хотим разделить сетку по

рядом. Для этого каждая строка будет иметь свой собственный цвет. На изображении ниже вы можете увидеть, как каждая группа процессов с одним цветом слева находится в своем собственном коммуникаторе справа.

Split a Large Communicator Into Smaller Communicators



Рассмотрим программный код, выполняющий эти действия:

```
MPI_Init(NULL, NULL);

int world_rank, world_size;

MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4;

MPI_Comm row_comm;

MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;

MPI_Comm_rank(row_comm, &row_rank);

MPI_Comm_size(row_comm, &row_size);
```

```
printf("WORLD RANK/SIZE: %d/%d --- ROW RANK/SIZE: %d/%d\n",  
world_rank, world_size, row_rank, row_size);
```

```
MPI_Comm_free(&row_comm);
```

```
MPI_Finalize();
```

Первые несколько строк получают ранг и размер исходного коммуникатора **MPI_COMM_WORLD**. Следующая строка делает важную операцию определения «цвета» локального процесса. Помните, что цвет определяет, к какому коммуникатору процесс будет принадлежать после разделения. Мы используем исходный ранг (**world_rank**) как ключ для операции разделения. Поскольку мы хотим, чтобы все процессы в новом коммуникаторе были в том же порядке, что и в оригинальном коммуникаторе, использование первоначального значения ранга здесь имеет наибольший смысл, поскольку оно уже будет правильно упорядочено. После этого мы печатаем новый ранг и размер, чтобы убедиться, что он работает. Результат выполнения программы должен выглядеть примерно так:

```
WORLD RANK/SIZE: 1/8 --- ROW RANK/SIZE: 1/4
```

```
WORLD RANK/SIZE: 0/8 --- ROW RANK/SIZE: 0/4
```

```
WORLD RANK/SIZE: 2/8 --- ROW RANK/SIZE: 2/4
```

```
WORLD RANK/SIZE: 7/8 --- ROW RANK/SIZE: 3/4
```

```
WORLD RANK/SIZE: 6/8 --- ROW RANK/SIZE: 2/4
```

```
WORLD RANK/SIZE: 5/8 --- ROW RANK/SIZE: 1/4
```

```
WORLD RANK/SIZE: 3/8 --- ROW RANK/SIZE: 3/4
```

```
WORLD RANK/SIZE: 4/8 --- ROW RANK/SIZE: 0/4
```

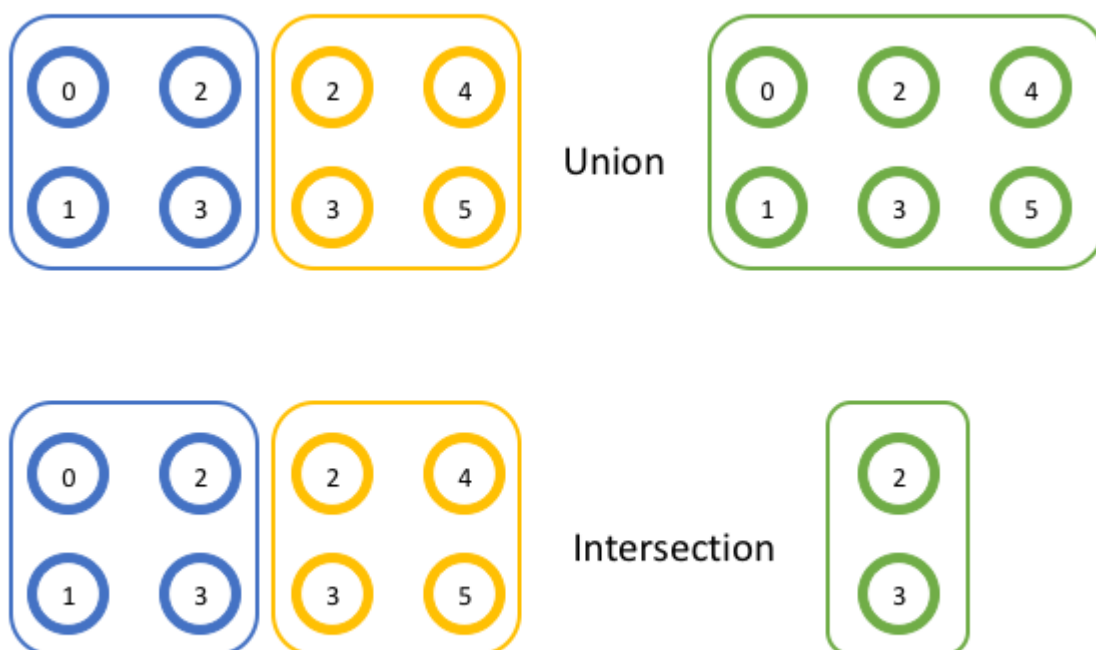
В конце программы мы освобождаем коммуникатор используя функцию **MPI_Comm_free**. Когда объект MPI больше не будет использоваться, он должен быть освобожден, чтобы впоследствии его можно было повторно использовать. MPI имеет ограниченное количество объектов, которые он может создавать одновременно. Если же не освобождать объекты, то это может привести к ошибке выполнения.

Хотя **MPI_Comm_split** это самая распространенная функция создания коммуникатора, есть много других. Например, **MPI_Comm_dup** создает дубликат коммуникатора. Может показаться странным, что существует функция, которая создает только копию, но это очень полезно для приложений, которые используют библиотеки для выполнения специализированных функций, таких как математические библиотеки. В таких приложениях важно, чтобы коды пользователей и коды библиотек не мешали друг другу. Чтобы этого избежать, первое, что должно делать каждое приложение, это создать дубликат **MPI_COMM_WORLD**, что позволит избежать проблемы с другими библиотеками, которые также используют **MPI_COMM_WORLD**. Сами же библиотеки должны делать дубликаты, **MPI_COMM_WORLD** чтобы избежать такой же проблемы.

Существуют и другие более сложные функции для работы с коммуникаторами, которые мы здесь не затрагиваем, такие как различия между меж-коммуникаторами и внутренними коммуникаторами и другие расширенные функции создания коммуникатора. Они используются только в очень специфических видах приложений.

Хотя **MPI_Comm_split** это самый простой способ создать новый коммуникатор, это не единственный способ сделать это. Существуют более гибкие способы создания коммуникаторов, но они используют новый тип объекта **MPI_Group**. Прежде чем подробно рассказывать о группах, давайте посмотрим немного больше на то, что коммуникатор на самом деле. Внутренне MPI должен поддерживать две основные части коммуникатора - контекст, который отличает один коммуникатор от другого и группу процессов, содержащихся в коммуникаторе. MPI хранит идентификатор для каждого коммуникатора внутри, чтобы предотвратить смешивание. Группу немного проще понять, поскольку это всего лишь совокупность всех процессов в коммуникаторе. Для **MPI_COMM_WORLD**, это все процессы, которые были начаты **mpirun**. Для других коммуникаторов группа будет отличаться.

MPI использует эти группы так же, как обычно работает теория множеств. Во-первых, операция объединения создает новый (потенциально) больший набор из двух других наборов. Новый набор включает в себя все члены первых двух наборов (без дубликатов). Во-вторых, операция пересечения создает новый (потенциально) меньший набор из двух других наборов. Новый набор включает в себя все элементы, которые присутствуют в обоих оригинальных наборах. Вы можете увидеть примеры обеих этих операций графически ниже.



В первом примере, объединение этих двух групп, $\{0, 1, 2, 3\}$ и $\{2, 3, 4, 5\}$ это $\{0, 1, 2, 3, 4, 5\}$ потому, что каждый из этих элементов появляется в каждой группе. Во втором примере, пересечение двух групп $\{0, 1, 2, 3\}$, и $\{2, 3, 4, 5\}$ это $\{2, 3\}$ потому, что только те элементы появляются в каждой группе.

Теперь, когда мы понимаем основы работы групп, давайте посмотрим, как они могут применяться к операциям MPI. В MPI легко получить группу процессов в коммуникаторе с вызовом API **MPI_Comm_group**.

MPI_Comm_group(MPI_Comm comm, MPI_Group* group)

Как упоминалось выше, коммуникатор содержит контекст, или идентификатор, и группу. Вызов **MPI_Comm_group** возвращает ссылку на этот объект группы. Объект группы работает так же, как объект-коммуникатор, за исключением того, что вы не можете использовать его для связи с другими рангами. Вы все равно можете получить ранг и размер для группы (**MPI_Group_rank** и **MPI_Group_size**, соответственно). Однако то, что вы можете делать с группами, которые вы не можете делать с коммуникаторами, - это использовать его для локального создания новых групп. Здесь важно запомнить разницу между локальной операцией и удаленной. Удаленная операция включает связь с другими рангами, где локальная операция не работает. Создание нового коммуникатора - это удаленная операция, потому что все процессы должны решать один и тот же контекст и группу, где создание группы является локальным, потому что оно не используется для связи и поэтому не обязательно должно иметь один и тот же контекст для каждого процесса. Вы можете манипулировать группой, которая вам нравится, без каких-либо сообщений. Когда у вас есть группа или две, выполнение операций довольно просто. Функция для получения объединения групп выглядит следующим образом:

```
MPI_Group_union( MPI_Group group1, MPI_Group group2, MPI_Group*  
newgroup)
```

Пересечение выглядит следующим образом:

```
MPI_Group_intersection( MPI_Group group1, MPI_Group group2,  
MPI_Group* newgroup)
```

В обоих случаях операция выполняется для **group1** и **group2**, а результат сохраняется **newgroup**.

8. ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ НА ОСНОВЕ OPENMP

8.1. Введение в OpenMP

OpenMP - механизм написания параллельных программ для систем с общей памятью. Состоит из набора директив компилятора и библиотечных функций. Позволяет достаточно легко создавать многопоточные приложения на C/C++, Fortran. Поддерживается производителями аппаратуры (Intel, HP, SGI, Sun, IBM), разработчиками компиляторов (Intel, Microsoft, KAI, PGI, PSR, APR, Absoft).

В программной модели OpenMP основной поток порождает дочерние потоки по мере необходимости. В модели **fork-join** программирование осуществляется путем вставки директив компилятора в ключевые места исходного кода программы. Компилятор интерпретирует эти директивы и вставляет в соответствующие места программы библиотечные вызовы для распараллеливания участков кода. Например:

Последовательный код

```
void main(){  
    double x[1000];  
    for(i=0; i<1000; i++){  
        calc smth(&x[i]);  
    }  
}
```

Параллельный код

```
void main(){  
    double x[1000];  
    #pragma omp parallel for ...  
    for(i=0; i<1000; i++){  
        calc smth(&x[i]);  
    }  
}
```

Директива **#pragma omp parallel for** указывает на то, что данный цикл следует разделить по итерациям между потоками.

Количество потоков можно контролировать из программы, или через среду выполнения программы-переменную окружения **OMP_NUM_THREADS**.

Следует отметить, что разработчик ответственен за синхронизацию потоков и зависимость между данными. Для того, чтобы скомпилировать программу с поддержкой OpenMP компилятору следует указать дополнительный ключ:

icc –openmp prog.c

ifc –openmp prog.f

В модели с разделяемой памятью взаимодействие потоков происходит через разделяемые переменные. При неаккуратном обращении с такими переменными в программе могут возникнуть ошибки соревнования (**race condition**). Такое происходит из-за того, что потоки выполняются параллельно и соответственно последовательность доступа к разделяемым переменным может быть различна от одного запуска программы к другому.

Для контроля ошибок соревнования работу потоков необходимо синхронизировать. Для этого используются такие примитивы синхронизации как критические секции, барьеры, атомарные операции и блокировки. Стоит отметить, что синхронизация может потребовать от программы дополнительных накладных расходов и лучше подумать, и распределить данные таким образом, чтобы количество точек синхронизации было минимизировано.

8.2. Основы OpenMP

Синтаксис. В основном, конструкции OpenMP - это директивы компилятора. Для C/C++ директивы имеют следующий вид:

#pragma omp конструкция [условие [условие]...]

Поскольку конструкции OpenMP являются директивами, то тот компилятор, который их не понимает, пропустит их и все же соберет OpenMP программу, правда последовательную.

В большинстве своем директивы OpenMP применимы только к структурным блокам, которые имеют единственную точку входа и единственную точку выхода. Единственным исключением является оператор STOP в языке Fortran и функция exit() в C/C++.

Правильно

```
#pragma omp parallel
{
L1:
wrk[id] = junk[id]; res[id] = wrk[id]*1.342; if(conv(res)) goto L1;
}
printf("%d", id);
```

Неправильно

```
{
L1:
    wrk[id] = junk[id];
L2:
    res[id] = wrk[id]*1.342;
    if(conv(res)) goto L3;
    goto L1;
}
if(not_done) goto L2;
L3:
    printf("%d", id);
```

8.3. Параллельные регионы

Параллельные регионы являются основным понятием в OpenMP. Именно там, где задан этот регион программа выполняется параллельно. Как только компилятор встречает прагму **omp parallel**, он вставляет инструкции для создания параллельных потоков.

Выше уже упоминалось, что количество порождаемых потоков для параллельных областей контролируется через переменную окружения **OMP_NUM_THREADS**, а также может задаваться через вызов функции внутри программы.

Каждый порожденный поток исполняет блок код в структурном блоке. По умолчанию синхронизация между потоками отсутствует и поэтому последовательность выполнения конкретного оператора различными потоками не определена.

После выполнения параллельного участка кода все потоки, кроме основного завершаются, и только основной поток продолжает исполняться, но уже один.

Каждый поток имеет свой уникальный номер, который изменяется от 0 (для основного потока) до количества потоков - 1. Идентификатор потока может быть определен с помощью функции **omp_get_thread_num()**.

Зная идентификатор потока, можно внутри области параллельного исполнения направить потоки по разным ветвям.

```
#pragma omp parallel  
{  
  myid = omp_get_thread_num(); if(myid == 0)  
    do_something();  
  else  
    do_something_else(myid);  
}
```

Приведенный пример обладает следующим недостатком. Мы не знаем является ли переменная **myid** разделяемой или приватной.

Область параллельного исполнения описывается в C/C++ следующим образом:

```
#pragma omp parallel \  
shared(var1, var2, ....) \
```

```

private(var1, var2, ...) \
firstprivate(var1, var2, ...) \
reduction(оператор:var1,var2, ...)
if(выражение) \
default(shared/none)
{
структурный блок
}

```

Существует две модели исполнения: динамическая, когда количество используемых потоков в программе может варьироваться от одной области параллельного выполнения к другой, и статическая, когда количество потоков фиксировано.

Модель исполнения контролируется или через переменную окружения **OMP_DYNAMIC** или с помощью вызова функции **omp_set_dynamic()**.

8.4. Конструкции OpenMP

Условия выполнения. Условия выполнения определяют то, как будет выполняться параллельный участок кода и область видимости переменных внутри этого участка кода. Опишем следующие условия:

shared(var1, var2,)

Условие **shared** указывает на то, что все перечисленные переменные будут разделяться между потоками. Все потоки будут иметь доступ к одной и той же области памяти.

private(var1, var2, ...)

Условие **private** указывает на то, что каждый поток должен иметь свою копию переменной на всем протяжении своего исполнения.

firstprivate(var1, var2, ...)

Это условие аналогично условию **private** за тем исключением, что указанные переменные инициализируются при входе в параллельный участок кода значением, которое имела переменная до входа в параллельную секцию.

lastprivate(var1, var2, ...)

Приватные переменные сохраняют свое значение, которое они получили при достижении конца параллельного участка кода.

reduction(оператор:var1, var2, ...)

Это условие гарантирует безопасное выполнение операций редукции, например, вычисление глобальной суммы.

if(выражение)

Это условие говорит о том, что параллельное выполнение необходимо только если выражение истинно.

default(shared|private|none)

Это условие определяет область видимости переменных внутри параллельного участка кода по умолчанию.

schedule(type[,chunk])

Этим условием контролируется то, как итерации цикла распределяются между потоками.

Условия private, shared, default. Рассмотрим следующие примеры:

```
#pragma omp parallel shared(a) private(myid, x)
{
    myid = omp_get_thread_num();
    x = work(myid);
    if(x < 1.0)
        a[myid] = x;
}

#pragma omp parallel default(private) shared(a)
{
    myid = omp_get_thread_num();
    x = work(myid);
    if(x < 1.0)
        a[myid] = x;
```

```
}
```

В обоих примерах каждый поток будет иметь свою копию переменных `x` и `myid`. Если эти переменные не будут объявлены как приватные, то их значение во время выполнения будет не определено. Значение переменных `x` и `myid` при входе в параллельный участок кода не определено и требуется инициализация этих переменных.

Во втором примере условие **default** автоматически указывает компилятору на необходимость завести свои переменные `x` и `myid` для каждого потока.

Условие **shared** в примерах говорит о том, что массив `a` является разделяемым между потоками и его значение сохраняется при выходе из параллельного участка кода.

Условие firstprivate. Переменные, которые подпадают под это условие являются приватными для каждого потока, но перед выполнением потока происходит их инициализация значением, которое было получено в предыдущем последовательном коде. Так в следующем примере до входа в параллельный участок кода значение переменной `a` равнялось **10**. Это же значение имеет эта переменная и при входе в параллельный участок кода.

```
int myid, a;  
a = 10;  
#pragma omp parallel default(private) \  
firstprivate(a)  
{  
    myid = omp_get_thread_num();  
    printf("Thread%d: a = %d\n", myid, a);  
    a = myid;  
    printf("Thread%d: a = %d\n", myid, a);  
}
```

Вывод

Thread1: a = 10

Thread1: a = 1

Thread2: a = 10

Thread0: a = 10

Thread3: a = 10

Thread3: a = 3

Thread2: a = 2

Thread0: a = 0

8.5. Конструкции OpenMP для распределения работ

Использование других условий более наглядно будет продемонстрировано на примере конструкций разделения работ. Таких конструкций всего три:

- параллельный цикл `for/DO`
- параллельные секции (sections)
- Конструкция `single`

Параллельный цикл `for/DO`. Цель конструкции - распределение итераций цикла по потокам.

```
#pragma omp parallel
{
    #pragma omp for private(i) shared(a,b)
    for(i=0; i<10000; i++)
        a[i] = a[i] + b[i]
}
```

По умолчанию барьером для потоков является конец цикла. Все потоки достигнув конца цикла дожидаются тех, кто еще не завершился, после чего основная нить продолжает выполняться дальше. Используя условие **nowait** для цикла можно разрешить основной нити не дожидаться завершения дочерних нитей.

Директива параллельного цикла `for/DO` имеет следующий синтаксис: Для C/C++:

#pragma omp for [условие [,условие] ...]

цикл for

где *условие* – это одно из:

private(var1, var2, ...)

shared(var1, var2, ...)

firstprivate(var1, var2, ...)

lastprivate(var1, var2, ...)

reduction(онепатомор: var1, var2,)

ordered

schedule(mun [, размер блока])

nowait

if(выражение)

Параллельные секции. Порой возникает необходимость параллельно выполнить действия, которые не являются итерациями цикла. Конечно можно воспользоваться для этих целей простой директивой **parallel**, но тогда придется писать дополнительный код, чтобы различную работу распределить между потоками. Более просто эту задачу можно решить с помощью параллельных секций.

#pragma omp parallel sections

{

#pragma omp section

{

printf("T%d: foo\n", omp_get_thread_num());

}

#pragma omp section

{

printf("T%d: bar\n", omp_get_thread_num());

```
}  
} // omp sections
```

Каждая секция выполняется в отдельном потоке, что позволяет производить декомпозицию по коду. Точкой синхронизации является конец блока **sections**. В случае, когда необходимо чтобы основной поток не ждал завершения остальных потоков следует использовать условие **nowait**.

Синтаксис параллельных секций в C/C++

```
#pragma omp sections \  
[условие [, условие...]]  
{  
#pragma omp section  
структурный блок  
[#pragma omp section  
структурный блок  
...]  
}
```

где условие – это одно из:

```
private(var1, var2, ...)  
firstprivate(var1, var2, ...)  
lastprivate(var1, var2, ...)  
reduction(оператор: var1, var2, ....)  
nowait
```

Конструкция single. Если в параллельной секции требуется выполнить какое-либо действие и при этом это действие должно быть выполнено только одним потоком (например, подсчет промежуточного результата), то для этого идеально подходит конструкция **single**.

Синтаксис в C/C++:

```
#pragma omp single [условие [, условие ...]]  
структурный блок
```

где условие – это одно из:

private(var1, var2, ...)

firstprivate(var1, var2, ...)

nowait

Условия выполнения (2). Условие if. В тех случаях, когда накладные расходы на порождение потоков могут быть больше, чем выигрыш от распараллеливания, то необходимо воспользоваться условием **if**.

```
#pragma omp parallel
{
    #pragma omp for if(n>2000)
    {
        for(i=0; i<n; i++)
            a[i] = work(i);
    }
}
```

В приведенном примере цикл будет распараллелен при условии, что итераций цикла больше, чем 2000.

Условие lastprivate. Это условие действует аналогично условию **private** за тем исключением, что значение переменной, вычисленное на последней итерации цикла, сохраняется.

```
#pragma omp parallel
{
    #pragma omp for private(i) lastprivate(k)
    for(i=0; i<10; i++)
        k = i*i;
}
printf("k = %d\n", k);
```


При выходе из цикла значение переменной *k* будет равно 100. Если бы переменная *k* была объявлена как приватная, то ее значение при выходе из цикла будет не определено.

Условие *reduction*. Это условие позволяет производить безопасное глобальное вычисление. Приватная копия каждой перечисленной переменной инициализируется при входе в параллельную секцию в соответствии с указанным оператором (0 для оператора +). При выходе из параллельной секции из частично вычисленных значений вычисляется результирующее и передается в основной поток.

```
#pragma omp parallel
{
    #pragma for shared(x) private(i) reduction(+:sum)
    for(i=0; i<100000; i++)
        sum += x[i];
}

#pragma omp parallel
{
    #pragma for shared(x) private(i) reduction(min:gsum)
    for(i=0; i<100000; i++)
        gmin = min(gmin, x[i]);
}
```

В C/C++ доступны следующие операторы и агрегатные функции: +, -, *, &, ^, |, &&, ||, min, max.

Условие *schedule*. Данное условие контролирует то, как работа будет распределяться между потоками.

schedule(тип [, размер блока])

Данное условие контролирует то, как работа будет распределяться между потоками.

schedule(тип [, размер блока])

Размер блока задает размер каждого пакета на обработку потоком (количество итераций).

Тип расписания может принимать следующие значения:

- **static** - итерации равномерно распределяются по потокам. Т.е. если в цикле 1000 итераций и 4 потока, то один поток обрабатывает все итерации с 1 по 250, второй - с 251 по 500, третий - с 501 по 750, четвертый с 751 по 1000. Если при этом задан еще и размер блока, то все итерации блоками заданного размера циклически распределяются между потоками. Статическое распределение работы эффективно, когда время выполнения итераций равно, или приблизительно равно. Если это не так, то разумно использовать следующий тип распределения работ.
- **dynamic** - работа распределяется пакетами заданного размера (по умолчанию размер равен 1) между потоками. Как только какой-либо из потоков заканчивает обработку своей порции данных, он захватывает следующую. Стоит отметить, что при этом подходе несколько большие накладные расходы, но можно добиться лучшей балансировки загрузки между потоками.
- **guided** - данный тип распределения работы аналогичен предыдущему, за тем исключением, что размер блока изменяется динамически в зависимости от того, сколько необработанных итераций осталось. Размер блока постепенно уменьшается вплоть до указанного значения. При таком подходе можно достичь хорошей балансировки при меньших накладных расходах.
- **runtime** - тип распределения определяется в момент выполнения программы. Это удобно в экспериментальных целях для выбора

Тип распределения работ зависит от переменной окружения **OMP_SCHEDULE**. По умолчанию считается, что установлен статический метод распределения работ.

```
bash> export OMP_SCHEDULE=static,1000
```

```
bash> export OMP_SCHEDULE=dynamic
```

ordering (упорядочивание). Порядок, в котором будут обрабатываться итерации цикла, вообще говоря, непредсказуем. Тем не менее, возможно «заставить» OpenMP выполнять выражения в цикле по порядку. Для этого существует ключевое слово **ordered**:

```
#pragma omp for ordered schedule(dynamic)  
for(int n=0; n<100; ++n)  
{  
    files[n].compress();  
    #pragma omp ordered  
    send(files[n]);  
}
```

Цикл «сжимает» 100 файлов в параллельном режиме, но «посылает» их строго в последовательном порядке. Если, например, поток «сжал» седьмой файл, но шестой файл к этому моменту ещё не был «отправлен», поток будет ожидать «отправки» шестого файла. Каждый файл «сжимается» и «посылается» один раз, но «сжатие» может происходить в параллельном режиме. Разрешено использовать только один **ordered** блок на цикл.

Переменные окружения OpenMP.

OMP_NUM_THREADS

Устанавливает количество потоков в параллельном блоке. По умолчанию, количество потоков равно количеству виртуальных процессоров.

OMP_SCHEDULE

Устанавливает тип распределения работ в параллельных циклах с типом `runtime`.

OMP_DYNAMIC

Разрешает или запрещает динамическое изменение количества потоков, которые реально используются для вычислений (в зависимости от загрузки системы). Значение по умолчанию зависит от реализации.

OMP_NESTED

Разрешает или запрещает вложенный параллелизм (распараллеливание вложенных циклов). По умолчанию - запрещено.

Библиотечные функции OpenMP. Для эффективного использования процессорного времени компьютера и написания гибких OpenMP программ пользователю предоставляется возможность управлять ходом выполнения программы посредством библиотечных функций. Библиотека OpenMP предоставляет пользователю следующий набор функций:

void omp_set_num_threads(int num_threads)

Устанавливает количество потоков, которое может быть запрошено для параллельного блока.

int omp_get_num_threads()

Возвращает количество потоков в текущей команде параллельных потоков.

int omp_get_max_threads()

Возвращает максимальное количество потоков, которое может быть установлено `omp_set_num_threads`.

int omp_get_thread_num()

Возвращает номер потока в команде (целое число от 0 до количества потоков - 1).

int omp_get_num_procs()

Возвращает количество физических процессоров доступных программе.

int omp_in_parallel()

Возвращает не нулевое значение, если вызвана внутри параллельного блока. В противном случае возвращается 0.

void omp_set_dynamic(expr)

Разрешает/запрещает динамическое выделение потоков.

int omp_get_dynamic()

Возвращает разрешено или запрещено динамическое выделение потоков.

void omp_set_nested(expr)

Разрешает/запрещает вложенный параллелизм.

int omp_get_nested()

Возвращает разрешен или запрещен вложенный параллелизм.

Перед использованием функций в фортране следует из объявить как соответствующий тип данных, в C/C++ - подключить файл заголовков `omp.h`.

```
#include <omp.h>
```

Изменения, сделанные функциями, являются приоритетнее, чем соответствующие переменные окружения. Так, функция **omp_set_num_threads()** переписывает значение переменной окружения **OMP_NUM_THREADS**, которое может быть установлено перед запуском программы.

8.6. Зависимость по данным в OpenMP

Для того, чтобы цикл мог быть распараллелен, работа, которая выполняется на одной итерации цикла не должна зависеть от работы на другой итерации. Другими словами, итерации цикла должны быть независимыми. Порой от зависимости по данным можно избавиться слегка переписав код:

```
for(i=1; i<8; i++)
```

```
  a[i] = c*a[i-1];
```

Здесь зависимость есть.

```
for(i=1; i<9; i+=2)
```

```
  a[i] = c*a[i-1];
```

Зависимости нет.

Утверждение 1

Только те переменные, в которые происходит запись на одной итерации и чтение их значения на другой создают зависимость по данным.

Утверждение 2

Только разделяемые переменные могут создавать зависимость по данным.

Следствие. Если переменная не объявлена как приватная, она может оказаться разделяемой и привести к зависимости по данным.

```
for(i=0; i<1000; i++){  
    x = cos(a[i]);  
    b[i] = sqrt(x*c);  
}
```

Вызовы функций внутри циклов – обычное дело. Однако и такие циклы могут быть распараллелены. Для этого программист должен сделать функцию независимой от внешних данных кроме как от значения параметров. В функции так же не должно быть статических переменных (**static**).

```
double foo(double *a, double *b, int i){  
    // Зависимость есть  
    ...  
    return 0.345*(a[i] + b[2*i]*C);  
}  
double bar(double a, double b){  
    // Зависимости нет  
    return 0.345*(a + b*C);  
}
```

Иногда возникают ситуации, когда индексы одного массива приходится хранить в другом массиве.

```
for(i=0; i<N; i++){  
    b[i] = c*a[indx1[i]];  
}  
for(i=0; i<N; i++){  
    b[indx2[i]] = sqrt(a[i]);  
}
```

В приведенном выше примере если `indx1[i]` не равен `i` на каждой итерации, то есть зависимость по данным. Если в массиве `indx2` есть повторения, то в цикле есть зависимость итераций по данным.

Циклы, в которых есть выход по условию не должны подвергаться распараллеливанию, поскольку эти циклы требуют упорядоченного выполнения.

```
for(i=0; i<1000; i++){  
    b[i] = sqrt(cos(a[i])*c);  
    if(b[i]>epsilon)  
        break;  
}
```

Рассмотрим еще один пример:

```
for(k=0; k<N; k++){  
    for(i=0; i<N; i++){  
        for(j=0; j<N; j++){  
            a[i][j] += b[i][k]*c[k,j];  
        }  
    }  
}
```

Если внешний цикл распараллелить, то получается зависимость по данным – `a[i][j]`. Для исправления такого положения вещей цикл по `k` следует сделать внутренним.

```
for(i=0; i<N; i++){  
    for(j=0; j<N; j++){  
        for(k=0; k<N; k++){  
            a[i][j] += b[i][k]*c[k,j];  
        }  
    }  
}
```

Порой бывает трудно определить есть ли зависимость по данным в коде и тогда на помощь разработчику может прийти компилятор.

8.7. Средства синхронизации в OpenMP

В OpenMP предусмотрены следующие конструкции синхронизации:

critical – критическая секция

atomic – атомарность операции

barrier – точка синхронизации

master – блок, который будет выполнен только основным потоком. Все остальные потоки пропустят этот блок. В конце блока неявной синхронизации нет.

ordered – выполнять блок в заданной последовательности

flush – немедленный сброс значений разделяемых переменных в память.

Критическая секция. Наличие критической секции в параллельном блоке гарантирует, что она в каждый конкретный момент времени будет выполняться только одним потоком. Т.е. когда один поток находится в критической секции, все остальные потоки, которые готовы в нее войти, находятся в приостановленном состоянии. Критические секции могут снабжаться именами. При этом критические секции считаются независимыми, только если они используют разные имена. По умолчанию, все непоименованные критические секции имеют одно имя.

Синтаксис критической секции на C/C++:

#pragma omp critical [(имя)]

Структурный блок

Пример (некорректное использование).

```
#pragma omp parallel for private(i) shared(a,xmax)
```

```
for(i=0; i<N; i++){
```

```
if(a[i]>xmax)
```

```
#pragma omp critical
```

```
xmax = a[i];
```

```
}// for
```

Пример (корректное использование, но не эффективное)

```
#pragma omp parallel for private(i) shared(a,xmax)
```

```
for(i=0; i<N; i++){
```

```
#pragma omp critical
```



```

if(a[i]>xmax)
xmax = a[i];
} // for

```

Атомарная секция. Барьеры. Барьеры – такой элемент синхронизации, который приостанавливает дальнейшее выполнение программы до тех пор, пока все потоки не достигнут этого барьера. Как только барьер достигнут всеми потоками, выполнение программы продолжается.

Синтаксис на C/C++

```
#pragma omp barrier
```

Или:

```

#pragma omp parallel
{
<инициализация>
#pragma omp barrier
<работа>
}

```

Фиксация порядка выполнения. Директива **ordered** в параллельных циклах (только там она может встречаться) говорит о том, что указанный блок должен исполняться в строго фиксированной последовательности. Внутри **ordered** секции одновременно может находиться только один поток.

Синтаксис на C/C++

```
#pragma omp ordered
```

Структурный блок

Пример.

```

#pragma omp parallel private(myid)
{
myid = omp_get_thread_num();
#pragma omp for private(i)
for(i=0; i<8; i++)

```

```
#pragma omp ordered
printf("T%d: %d\n", myid, i);
}
```

Результат работы кода следующий:

T0: 0

T0: 1

T0: 2

T0: 3

T1: 4

T1: 5

T1: 6

T1: 7

Конструкция flush. Эта конструкция осуществляет немедленный сброс значений разделяемых переменных в память. Таким образом гарантируется, что во всех потоках значение переменной будет одинаковое. Неявно **flush** присутствует в следующих директивах: **barrier**, начале и конце критических секций, параллельных циклов, параллельных областей, **single** секций.

С ее помощью можно посылать сигналы потоком используя переменную как семафор. Когда поток видит, что значение разделяемой переменной изменилось, то это говорит, что произошло событие и, следовательно, можно продолжить выполнение программы далее. (Пример не работает. Не происходит блокирования)

Синтаксис:

```
#pragma omp flush(var1[, var2, ...])
```

8.8. Расширенные возможности OpenMP

Рассмотрим средства OpenMP, которые позволяют более эффективно писать параллельные программы. К таким средствам относится директива **threadprivate**, которая позволяет один раз объявить приватную переменную для всех параллельных секций в рамках одного файла. Чтобы было возможно ее

использовать, переменная должна быть объявлена как статическая, директива **threadprivate** должна присутствовать до объявления первой параллельной секции и количество потоков в программе должно быть постоянным.

Синтаксис:

```
#pragma omp threadprivate(var1[, var2 ...])
```

Другим расширением OpenMP является возможность использовать синхронизацию потоков посредством блокировок. Блокировки в OpenMP аналогичны мютексам в POSIX threads. Даже набор функций для работы с ними аналогичен.

void omp_init_lock(omp_lock_t *lock)

инициализирует блокировку и связывает ее с параметром lock.

void omp_destroy_lock(omp_lock_t *lock)

деинициализирует переменную, связанную с параметром lock.

void omp_set_lock(omp_lock_t *lock)

Блокирует выполнение потока до тех пор, пока блокировка на переменную lock не станет доступной.

void omp_unset_lock(omp_lock_t *lock)

Снимает блокировку с переменной lock.

void omp_test_lock(omp_lock_t *lock)

Пытается установить блокировку и, если операция выполнена успешно, возвращает не нулевое значение. В противном случае возвращается ноль. Функция не блокирующая.

Прототипы функций описаны в omp.h

...

```
#include <omp.h>
```

```
void main(){
```

```
omp_lock_t lock;
```

```
int i, p_sum = 0, res = 0;
```

```
omp_init_lock(&lock);
```

```

#pragma omp parallel firstprivate(p_sum)
{
#pragma parallel for private(i)
for(i=0; i<100000; i++)
p_sum +=i;
omp_set_lock(&lock);
res += p_sum;
omp_unset_lock(&lock);
}
omp_destroy_lock(&lock);
printf("%d\n", res);
}

```

8.9. Отладка OpenMP кода

Настройка производительности OpenMP кода. Будем считать, что вопрос о том стоит ли оптимизировать программу или нет не стоит. Предположим, что вас не устраивает производительность и Вы решили заняться оптимизацией.

Оптимизацию любой программы стоит начинать уже на стадии выбора алгоритма поскольку именно за счет правильно выбранного алгоритма можно получить прирост производительности на порядки. Чуть менее значимый вклад дает оптимизация реализации и распараллеливание. В рамках текущего курса - это использование OpenMP. Поэтому прежде чем распараллеливать программу с OpenMP настоятельно рекомендуется добиться максимальной производительности последовательной версии.

Основной подход

Исходя из общих соображений можно предложить следующий подход:

- Использовать автоматическое распараллеливание средствами компилятора.

- С помощью профилировщика выявить участки кода, которые наиболее требовательны к процессорному времени.
- Добавить директивы OpenMP для наиболее важных циклов.
- Если такое распараллеливание не дало ожидаемого прироста производительности, то выполнить проверку на
 - стоимости порождения процессов
 - размер циклов
 - балансировку загрузки
 - количество ссылок на разделяемые переменные
 - излишнюю синхронизацию
 - стоимость доступа к памяти

Рассмотрим некоторые моменты более подробно.

Автоматическое распараллеливание. Многие компиляторы, которые поддерживают OpenMP позволяют производить автоматическое распараллеливание программ. При этом распараллелены могут быть только циклы, в которых компилятор не нашел зависимости по итерациям. Анализируя код компилятор сам вставляет в программу директивы OpenMP. В случае успеха пользователь уведомляется о том, что цикл был распараллелен, если нет, то сообщается почему.

```
$> icc -parallel -par_report3 text.c
```

```
test.c(61) : (col. 5) remark: LOOP WAS AUTO-PARALLELIZED.
```

```
parallel loop: line 61
```

```
shared: {"A", "B"}
```

```
private: {"i", "j"}
```

```
first private: { }
```

```
reductions: { }
```

```
procedure: chk_bk
```

```
serial loop: line 74: not a parallel candidate due to insufficient work
```

```
serial loop: line 82: not a parallel candidate due to insufficient work
```

serial loop: line 66

anti data dependence assumed from line 68 to line 68, due to "B"

Здесь для компилятора Intel ключ `-parallel` указывает на необходимость выполнит автоматическое распараллеливание, ключ `-par_report` говорит о том, что необходимо выводить отчетность по распараллеливанию. Число в конце ключа – уровень отчетности.

Иногда компилятор может предполагать наличие зависимости по данным в цикле, которых реально нет, и как результат, отказываться от его распараллеливания. В этих случаях компилятору можно помочь, указав, что в этом цикле итерации независимы.

Профилирование программы. Иерархия памяти. Большинство вычислительных систем имеют следующую иерархию памяти:

1. регистры
2. кэш первого уровня
3. кэш второго уровня
4. локальная память
5. удаленная память (например, память другого узла кластера или жесткий диск)

При этом чем ниже по списку, тем большее время требуется на извлечение данных из соответствующей памяти.

Следовательно, в целях оптимизации надо более эффективно использовать локальную память, кэш и минимизировать обращения к удаленной памяти. Для этого надо стараться размещать данные в памяти так, чтобы обращение к ним происходило с минимальным количеством переписываний или вообще без переписывания кэша. Т.е. доступ к элементам массива должен осуществляться в той последовательности, в которой они лежат в памяти. Так, при работе с многомерными массивами в C/C++ наиболее быстро будет происходить доступ к элементам по самому правому (по записи)

индексу, а в Фортране по самому левому. Иногда, для оптимизации работы с памятью следует поменять местами вложенные циклы.

```
for(j=1; j<M-1; j++)
for(i=1; i<N-1; i++)
  B[i][j] = (A[i][j-1] + A[i][j+1] + A[i-1][j] + A[i+1][j])/4.0;

for(i=1; i<N-1; i++)
for(j=1; j<M-1; j++)
  B[i][j] = (A[i][j-1] + A[i][j+1] + A[i-1][j] + A[i+1][j])/4.0;
```

9. Применение Windows API в параллельных вычислениях

В контексте исполнения процесса могут выполняться несколько потоков. В операционной системе Windows поток – это единица исполнения, которой ОС выделяет процессорное время для выполнения программы.

Рассмотрим функций доступные в Windows API для работы с потоками. Для того, чтобы создать поток используется функция **CreateThread**.

```
HANDLE WINAPI CreateThread(
    LPSECURITY_ATTRIBUTES lpSecAttr,
    SIZE_T StackSize,
    LPTHREAD_START_ROUTINE lpStartFuncAddr,
    LPVOID p,
    DWORD dwCreatParam,
    LPDWORD thrId
);
```

где:

- *lpSecAttr* – указатель на SECURITY_ATTRIBUTES, этот параметр также можно указать равным NULL, тогда возвращаемый дескриптор не будет наследоваться
- *StackSize* – начальный размер стека (указывается в байтах). Если в качестве параметра указать ноль, то система задаст размер стека по умолчанию

- *lpStartFuncAddr* – это указатель на функцию, которая будет исполняться потоком, таким образом здесь будет указан стартовый адрес потока; эта функция должна быть определена в программе следующим образом:
DWORD WINAPI FunctionName(LPVOID)
- *p* – это указатель на переменную, которую нам необходимо передать в функцию *FunctionName(LPVOID p)*, исполняемую потоком
- *dwCreatParam* – параметры, которые управляют созданием потока. Можно указать: 0 (тогда поток начнет исполняться сразу после его создания), **CREATE_SUSPENDED** (поток начнет исполняться, когда будет выполнена функция *ResumeThread*, о ней позже)
- *thrId* – это указатель на переменную, в которую будет записан идентификатор потока

ExitThread – эта функция завершает поток, как видно из её определения ниже, она ничего не возвращает. Поток может завершиться при вызове этой функции или при возврате из функции **DWORD WINAPI FunctionName(LPVOID)**, которая исполнялась в потоке.

VOID WINAPI ExitThread(DWORD dwExitCode);

- в *dwExitCode* помещаем код завершения потока

Чтобы приостановить поток, нужно использовать функцию – *SuspendThread*. В случае успешной остановки потока функция вернёт предыдущее число остановок данного потока, есть исполнение функции *SuspendThread* завершится неудачей, то она вернёт -1.

DWORD WINAPI SuspendThread(HANDLE thread);

- *thread* – дескриптор потока, который необходимо приостановить. Он должен иметь права доступа: **THREAD_SUSPEND_RESUME**

Если нужно возобновить данный поток, то необходимо вызвать функцию *ResumeThread*. При успешном выполнении возвращается предыдущее количество остановок потока, иначе -1.

DWORD WINAPI ResumeThread(HANDLE thread);

- *thread* – дескриптор потока, который необходимо возобновить. Он должен иметь права доступа: `THREAD_SUSPEND_RESUME`

Теперь рассмотрим пример программы, которая осуществляет параллельную многопоточную генерацию и обработку строк двумерной матрицы. То есть каждый отдельный поток будет генерировать строку матрицы и находить произведение нечётных элементов этой строки. В каждый поток будем передавать в качестве параметра указатель на структуру, в которой будет храниться строка матрицы и результат произведения элементов.

Подключим необходимые библиотеки: для вывода результатов на экран, для использования функций WinAPI и для использования времени в качестве инициализирующего элемента генератора случайных чисел.

```
#include <stdio.h>
```

```
#include <Windows.h>
```

```
#include <time.h>
```

Определим константу для хранения размера матрицы (в данном примере матрица размером 4×4).

```
#define MATRIX_SIZE 4
```

Определим структуру, в которой будем хранить строку матрицы, результат умножения нечётных элементов этой строки и случайное число, которым будет инициализирован генератор случайных чисел в данном потоке (если использовать один и тот же генератор для всех потоков, то, из-за одновременности выполнения, будут генерироваться одинаковые числа для всех строк).

```
struct row
```

```
{
```

```
    int value[MATRIX_SIZE];
```

```
    int result;
```

```
    int rnd;
```

```
};
```

Функция DWORD WINAPI **generateAndCalc(void *data)**, которую будут исполнять потоки (с комментариями):

```
DWORD WINAPI generateAndCalc(void *data)
{
    //преобразуем полученные данные к типу структуры
    row *r = (row *) data;
    //инициализируем генератор случайных чисел полученным числом
    srand(r->rnd);
    //генерируем элементы строки
    for (int i = 0; i < MATRIX_SIZE; i++)
    {
        r->value[i] = rand() % 10;
    }
    //находим произведение нечетных элементов
    r->result = 1;
    for (int i = 0; i < MATRIX_SIZE; i += 2)
    {
        r->result *= r->value[i];
    }
}
```

Функция *main()* с комментариями:

```
int main()
{
    //инициализируем генератор случайных чисел
    srand(time(NULL));
    //определяем дескрипторы потоков,
    //идентификаторы потоков и структуры для строк матрицы
    HANDLE thread[MATRIX_SIZE];
    DWORD thrId[MATRIX_SIZE];
```

```

row rows[MATRIX_SIZE];

for (int i = 0; i < MATRIX_SIZE; i++)
{
    //генерируем случайные числа для каждой строки
    rows[i].rnd = rand();

    //создаем потоки
    thread[i] = CreateThread(NULL, 0, &generateAndCalc, &rows[i], 0,
&thrId[i]);
}

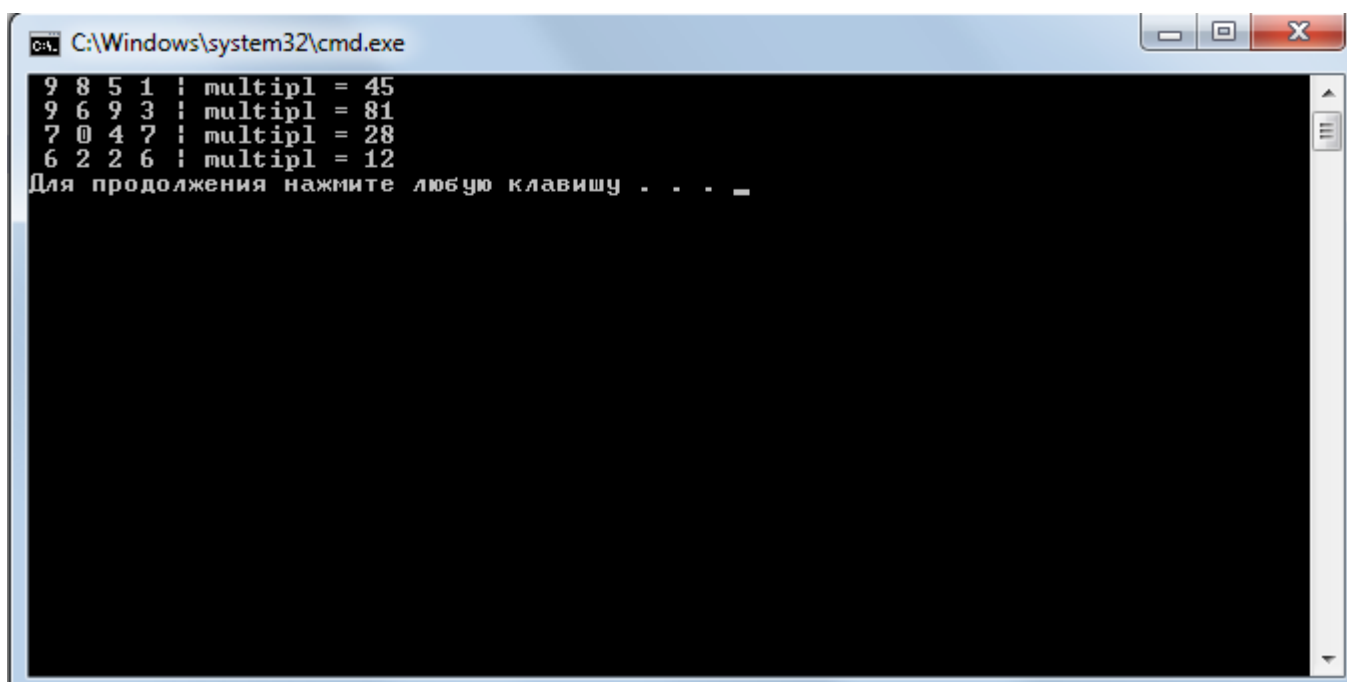
//ждем, пока все эти потоки завершатся
WaitForMultipleObjects(MATRIX_SIZE, thread, TRUE, INFINITE);

//выводим результат работы программы на экран
for (int i = 0; i < MATRIX_SIZE; i++)
{
    for (int j = 0; j < MATRIX_SIZE; j++)
    {
        printf(" %d", rows[i].value[j]);
    }
    printf(" / multipl = %d\n", rows[i].result);
}

return 0;
}

```

Работа программы демонстрируется на скриншоте ниже:

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\system32\cmd.exe'. The window contains the following text:

```
9 8 5 1 : multipl = 45
9 6 9 3 : multipl = 81
7 0 4 7 : multipl = 28
6 2 2 6 : multipl = 12
Для продолжения нажмите любую клавишу . . . _
```

Литература

1. Антонов А.С. Технологии параллельного программирования MPI и OpenMP: Учеб. пособие. Предисл.: В.А.Садовничий.-М.: Издательство Московского университета, 2012.
2. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. - Н.Новгород, ННГУ, 2001.
3. Богачев К.Ю. Основы параллельного программирования. - М.: БИНОМ. Лаборатория знаний, 2003.
4. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. - СПб.: БХВ-Петербург, 2002.
5. Немнюгин С., Стесик О. Параллельное программирование для многопроцессорных вычислительных систем - СПб.: БХВ-Петербург, 2002.
6. Корнеев В.В.. Параллельные вычислительные системы. - М.: Нолидж, 1999.
7. Корнеев В.В. Параллельное программирование в MPI. Москва-Ижевск: Институт компьютерных исследований, 2003.

8. Вагнер, Билл С# Эффективное программирование / Билл Вагнер. - М.: ЛОРИ, 2013. - 320 с.
9. Зиборов, В.В. Visual С# 2012 на примерах / В.В. Зиборов. - М.: БХВ-Петербург, 2013. - 480 с.
10. Подбельский, В. В. Язык С#. Базовый курс / В.В. Подбельский. - М.: Финансы и статистика, Инфра-М, 2011. - 384 с.
11. Прайс, Джейсон Visual С# 2.0. Полное руководство / Джейсон Прайс , Майк Гандэрлой. - М.: Век +, Корона-Век, Энтроп, 2010. - 736 с.
12. Рихтер, Джеффри CLR via С#. Программирование на платформе Microsoft .NET Framework 4.0 на языке С# / Джеффри Рихтер. - М.: Питер, 2013. - 928 с.
13. Троелсен, Эндрю Язык программирования С# 5.0 и платформа .NET 4.5 / Эндрю Троелсен. - М.: Вильямс, 2015. - 486 с.
14. <http://oi.ssau.ru/docs /lecparall.pdf>
15. <https://vscode.ru/prog-lessons/parallelnyie-vyichisleniya-s-pomoshhyu-winapi.html>